Remerciements

En premier lieu, je remercie le Bon Dieu de m'avoir donné la force et le courage pour accomplir ce travail et qui m'a procuré ce succès.

Je tiens à remercier toutes personnes, dont l'intervention au cours de cette mémoire à favorisé son aboutissement.

Je tiens à remercier Mr. ***Cherki DAOUI*** *mon encadrant à la FSTBM, pour son suivi et ses remarques avisées tout au long de mon projet, ses directives précieuses et conseils pertinents. Sa lecture minutieuse de ce document a beaucoup amélioré sa clarté. Puisse-t-il trouver ici l'expression de mon profond respect et de toute ma reconnaissance.*

Les expressions de notre vive reconnaissance s'adressent également à Mr. ***Mohammed FAKIR*** *le responsable du master Informatique Décisionnelle, Puisse-t-il trouver ici mes chaleureux remerciements pour ses précieux conseils et directives.*

Je tiens également à remercier vivement tout le corps professoral de la FSTBM pour la formation solide qu'il nous a assuré durant ces deux années d'études, tout en espérant que notre travail soit à la hauteur des attentes de nos professeurs.

Je remercie également tous les enseignants qui nous ont encadrés tout le long de ces deux années d'études en Master Informatique Décisionnelle.

Résumé

Les Processus Décisionnels de Markov (**PDM**), du fait de leur adaptation à la prise en compte d'incertitudes, sont étudiés depuis quelques dizaines d'années dans la communauté Intelligence Artificielle. Les stratégies obtenues sont donc beaucoup plus robustes aux incertitudes que ceux obtenus par les méthodes déterministes. D'autre part, avec l'augmentation de l'utilisation des **PDM,** dans plusieurs domaines, alors des autres contraintes s'apparaissent, parmi lesquelles, on trouve le problème de grandes tailles, la qualité des solutions et le temps nécessaires pour prendre la décision. C'est dans ce cadre que s'inscrit ce rapport de projet de fin d'étude dans lequel nous présentons une étude comparative, des solveurs **PDM**, dans laquelle nous avons fait, tout d'abord une comparaison théorique, pour assurer l'existence des solutions, des algorithmes étudier, puis, on a fait la comparaison selon la complexité de chaque algorithme, les modifications des algorithmes et les étapes clés qui peuvent rendre un algorithme plus performant que l'autres. L'objectif que nous avons, c'est de sortir avec une étude qu'on peut baser sur, afin de savoir quel est le meilleur algorithme à utiliser pour chaque problème, avec un choix qui se fait selon la taille du problème, le nombre d'actions et d'autres paramètres qu'on va les voir durant toutes cette étude.

Nous avons, proposé des versions parallèles de ces algorithmes, pour l'objectif de réduction de temps d'exécution, et les comparer après avec les algorithmes structurels, pour observer leurs efficacités. Nous n'avons pas arrêté à ce niveau, mais il fallait faire encore une analyse selon la nature de distribution, le nombre des processeurs à utiliser, vu que le partages des calcule entre les processeurs, va créer des opérations qui coutent aussi du temps.

Avec cette étude comparative, et les autres algorithmes parallèles qu'on a proposés nous avons utilisé tout ce travail, pour construire une solution qui combine entre ces algorithmes et les techniques existantes, comme la décomposition en niveaux, afin de sortir avec un nouvel algorithme efficace, pour l'appliquer sur les problèmes de grandes tailles. Dans ce nouvel algorithme nommé **PHVI**, nous avons profité à la fois des techniques de décompositions, les algorithmes parallèles qu'on a proposés et des calculs mathématiques, pour effectuer enfin, une distribution dynamique des processeurs, selon la taille de chaque partie du problème attaqué.

Abstract

Markov Decision-Making Processes (MDPs) have been studied for several decades in the Artificial Intelligence community because of their adaptation to taking into consideration uncertainties. The strategies obtained are therefore much more robust to uncertainties than those obtained by deterministic methods. On the other hand, with the increase of the use of ***MDP****s in several fields, other constraints appear, among which are the problem of large sizes, the quality of solutions and especially the size.*

It is within this framework that this research project report is carried out in which we present a comparative study, ***MDP*** *solvers, in which we made, first of all, a theoretical comparison, to ensure the existence of solutions, algorithms to study, then, we made the comparison according to the complexity of each algorithm, the modifications of the algorithms and the key steps that can make one algorithm more efficient than the others. Our objective is to come out with a study that we can base our work on so we can know what is the best algorithm to use for each problem, with a choice that is made according to the size of the problem, the number of actions and other parameters that we will see them during this study.*

We have proposed parallel algorithms, for the objective of reducing execution time, and then compare them with the structural algorithms, to observe their efficiency. We didn't stop at this level, but we still had to do an analysis according to the nature of the distribution, the number of processors to use, since the sharing of the calculations between the processors, will create operations that also time for decision making.

With this study that we have carried out and the other parallel algorithms that we have proposed, we have used all this work to build a solution that combines these algorithms with existing techniques, such as level decomposition, in order to come out with a new efficient algorithm to apply it on large scale problems. In this new algorithm called PHVI, we have taken advantage of both decomposition techniques, the proposed parallel algorithms and mathematical calculations to finally perform a dynamic distribution of processors, depending on the size of each part of the problem in hand.

Table de matières

Table des matières

Remerciements 2
Résumé 3
Abstract 3
Liste des tableaux 6
Liste des figures 7
Liste des algorithmes 8
Introduction générale 9
Chapitre I : Processus Décisionnel de Markov 11
1.1. Introduction 11
1.2. Processus stochastique 11
1.3. Chaine de Markov 11
1.4. PDM 12
1.4.1. Cadre théorique 12
1.4.2. Les politiques 13
1.4.3. Critère d'optimisation 13
1.4.4. Fonction de valeurs 14
1.4.5. Equations d'optimalité 14
1.5. Applications 16
1.6. Conclusion 16
Chapitre II : Algorithmes de résolution PDM sous le critère d'actualisation 18
2.1. Introduction 18
2.2. Algorithmes de résolution 18
1.5.1 Le critère fini 19
1.5.2. Le critère γ-pondéré 19
2.3. Programmation linéaire 24
2.2.1 Optimisation linéaire 24
2.2.2 Résolution d'un problème de la programmation linéaire 25
2.2.3 Méthodes de résolution d'un problème de programmation linéaire 26
2.2.4 Quelques problèmes de la programmation linéaire 30
2.4. Algorithme d'itération de la valeur et ses variantes 30
2.4.1. Algorithme d'itération de la valeur 31

2.4.2. Algorithme Gauss-Seidel Itération de la valeur 34
2.4.3. Conclusion partielle 36
2.5. Algorithme d'itération de la politique et ses variantes 36
2.5.1. Algorithme d'itération de la politique 37
2.6. Algorithmes parallèles 38
2.7. Conclusion 38
Chapitre III : Etude comparatives des solveurs 39
3.1. Introduction 39
3.2. Algorithme d'itération de la valeur et ses variantes 39
3.3. Algorithme d'itération de la politique et ses variantes 45
3.4. Algorithmes d'itération de la valeur et d'itération de la politique 47
3.5. Algorithmes parallèles 48
3.5.1. Le parallélisme 49
3.5.2. Algorithme parallèle d'itération de la valeur 50
3.5.3. Algorithme parallèle d'itération de la politique modifié 53
3.5.4. Comparaison des algorithmes parallèles 54
2.6. Conclusion 55
Chapitre V : Affectation dynamique des processeurs 56
4.1. Introduction 56
4.2. Composantes fortement connexes 56
4.3. Décomposition en niveaux 56
4.4. Distribution Dynamique 57
4.5. Algorithme 58
4.6. Comparaison des résultats 59
4.7. Conclusion 59
Conclusion générale & perspectives 60
Références : 61

Liste des tableaux

Tableau 1 : Les valeurs du vecteurs gains optimale V, de l'algorithme d'itération de la valeur dans chaque itération..... 33
Tableau 2 : Les valeurs du vecteurs gains optimale V, de l'algorithme d'itération de la politique dans chaque itération..... 35
Tableau 3 : Les valeurs de vecteur gain V et stratégies, optimales dans chaque itération, de l'algorithme MPI 45

Liste des figures

Figure 1 : Algorithme d'induction rétroactif pour le critère fini. 18
Figure 2 : Algorithme d'induction rétroactif pour le critère fini 19
Figure 3 : Programmation linéaire, critère γ-pondéré 20
Figure 4 : Algorithme d'itération de la valeur, critère γ-pondéré 20
Figure 5 : Algorithme d'itération de la valeur, critère γ-pondéré, Gauss-Seidel 21
Figure 6 : Algorithme d'itération de la politique, critère γ-pondéré 22
Figure 7 : Algorithme d'itération de la politique modifié, critère γ-pondéré 23
Figure 8 : Organigramme de la méthode simplexe. 29
Figure 9 : Comparaison entre l'algorithme d'itération de la valeur et Gauss-Seidel 40
Figure 10 : Comparaison entre l'algorithme d'itération de la valeur et Gauss-Seidel avec un nombre d'action plus grand 41
Figure 11 : Comparaison au niveau d'itération entre l'algorithme VI et GSVI 42
Figure 12 : Comparaison entre les algorithmes : VI, GS, VI_J et GS_J 43
Figure 13 : Etude de temps de convergence de l'algorithme VI selon Alpha sur un espace d'état E=50.000 44
Figure 14 : Etude de convergence selon le temps en changeant Alpha, sur un espace E=50.000 44
Figure 15 : Evolution selon le temps en fonction d'espace d'état, de l'algorithme PI 46
Figure 16 : Comparaison entre l'initialisation aléatoire et normal de la politique, avec l'algorithme MPI 47
Figure 17 : Comparaison au niveau de temps d'exécution de l'algorithme MPI et GSVI selon le nombre d'états 48
Figure 18 : Algorithme d'itération de la valeur parallèle 50
Figure 19 : Comparaison selon le temps d'exécution de l'algorithme VI structurel et VI parallèle..... 51
Figure 20 : Etude d'évolution du temps d'exécution suivant le nombre de threads 52
Figure 21 : Algorithme d'itération de la politique parallèle 53
Figure 22 : Comparaison entre les algorithmes d'itération de la politique structurel et parallèle..... 54
Figure 23 : Comparaison entre les algorithme parallèles, itération de la valeur et l'itération de la politique modifié 55
Figure 24 : Exemple d'un PDM, décomposé en niveaux 57
Figure 25 : Présentation de l'algorithme Parallèle Hiérarchique VI 58
Figure 26 : comparaison entre PHVI et la distribution classique 59

Liste des algorithmes

Algorithme 1 : Algorithme d'induction rétroactif pour le critère fini 18
Algorithme 2 : Algorithme d'induction rétroactif pour le critère fini 19
Algorithme 3 : Programmation linéaire, critère γ-pondéré 20
Algorithme 4 : Itération de la valeur, critère γ-pondéré 20
Algorithme 5 : Itération de la valeur, critère γ-pondéré, Gauss-Seidel 21
Algorithme 6 : Itération de la politique, critère γ-pondéré 22
Algorithme 7 : Itération de la politique modifié, critère γ-pondéré 23
Algorithme 8 : Itération de la valeur (Algorithme parallèle) 50
Algorithme 9 : Algorithme parallèle d'Itération de la politique modifié 53
Algorithme 10 : L'algorithme VI Parallèle Hiérarchique 58

Introduction générale

Dès les années 1950 émerge une classe de processus stochastiques contrôlés : les processus décisionnels de Markov (PDM). Il s'agit de suites de variables aléatoires (les états du système) dont la transition d'un état à un autre dépend de l'état de départ et d'une décision extérieure. Une quantité, coût ou récompense selon l'interprétation, mesure l'optimalité du choix opéré. Des critères doivent alors évaluer la qualité de toutes les décisions prises, dont l'ensemble constitue ce qu'on appelle une stratégie. Il existe de nombreux types de **PDM** : complétement ou partiellement observables, à temps discret ou à temps continu, à horizon fini ou infini. Un grand nombre de situations d'optimisation décisionnelle peuvent trouver un PDM adapté. Ainsi, on retrouve ce formalisme mathématique dans de nombreux domaines, notamment la finance, l'industrie ou la gestion. Même des situations de la vie quotidienne peuvent être modélisées par un PDM : par exemple, sous réserve d'une étude statistique préalable, on peut appliquer les PDM à la recherche d'une place dans un parking.

Les applications des **PDM** ont donné l'impulsion pour le développement de procédures d'optimisation et ont permis une littérature théorique et appliquée abondante sur le sujet. Ainsi, la réponse mathématique apportée à des problèmes concrets toujours plus nombreux et plus complexes implique d'élaborer des modèles de plus en plus précis.

Problématique : Les problèmes de grande taille et le cout.

En général, il existe beaucoup de problèmes, du monde réel qui sont très importants, et qui peuvent être modélisés sous forme d’un PDM et ainsi résolus par un algorithme de résolution des PDM. Mais un problème de taille s’impose, vu que les espaces d'état s’augmentent d’une façon exponentielle, ce qui fait que les complexités de ces solveurs sont au moins polynomiales. Par la suite, une augmentation de coût de la mémoire et du temps, alors ce n'est pas pratique d’appliquer sur ces problèmes de grande taille un algorithme qui est choisi aléatoirement, sans passer par des études ayant comme but de comparer ces solveurs, afin de choisir le meilleur pour chaque problème.

En conséquence, l’objectif principal de ce travail est de faire une étude comparative des solveurs PDM, afin de pouvoir sortir avec des résultats qui peuvent nous aider à décider quel algorithme à utiliser pour chaque problème ou situation, en se basant sur les paramètres liés à ces problèmes qui sont le nombre d’état, nombre d’action… Cette étude comprend aussi des analyses

au niveau des paramètres liés aux solveurs, afin de pouvoir faire un bon choix des paramètres initiaux, par exemple la valeur du seuil, qui contrôle la convergence des algorithmes.

Le mémoire est organisé comme suit :

Le premier chapitre présente des introductions générales sur les processus Stochastiques, les Chaînes de Markov et les Processus Décisionnels de Markov. Dans la partie des PDM, nous allons explorer le cadre théorique des PDM, notamment, les politiques, les critères d'optimisation, la fonction de valeur et enfin l'équation d'optimalité.

Le deuxième chapitre va commencer par une présentation des algorithmes de résolution des PDM, sous le critère d'actualisation, à savoir, la programmation linéaire, algorithme d'itération de la valeur avec ses variantes et l'algorithme d'itération de la politique avec ses variantes. Ces algorithmes seront étudiés en détails, au niveau des théorèmes de convergence et leurs preuves, pour passer à la fin, pour voir le parallélisme, afin de l'initialiser pour le chapitre qui vient.

Le troisième chapitre débute avec une introduction, pour définir l'environnement du travail, ainsi que les langages utilisés durant toute cette étude, puis on passe à la comparaison de tous les algorithmes, structurels, selon beaucoup des paramètres. Enfin, une comparaison des algorithmes parallèle est réalisée.

Dans le dernier chapitre, nous allons se baser sur toutes ces études faites aux chapitres précédents, pour construire un algorithme efficace pour la résolution des PDM de grandes tailles. Cet algorithme va profiter de l'algorithme parallèle, le plus performant, ainsi qu'une distribution dynamique des processeurs, afin de dépasser la distribution classique et d'améliorer la performance des calculs.

Chapitre I : Processus Décisionnel de Markov

1.1. Introduction

Un processus (problème) de décision markovien ($\boldsymbol{PDM}$) est un modèle stochastique issu de la théorie de la décision et de la théorie des probabilités. Le modèle $\boldsymbol{PDM}$ peut être vu comme une chaîne de Markov à laquelle on ajoute une composante décisionnelle et une fonction de récompense. Un $\boldsymbol{PDM}$ est un problème qui couple les deux problématiques de décision séquentielle et décision dans l'incertain, c'est une formalisation mathématique qui généralise les approches de plus court chemin dans un environnement stochastique.

Les $\boldsymbol{PDMs}$ sont donc, des chaînes de Markov visitant les états, contrôlés par les actions et évalués par des revenus. Résoudre un $\boldsymbol{PDM}$, c'est le contrôler de manière optimale. Les solutions d'un $\boldsymbol{PDM}$ sont des politiques, ou stratégies ou encore règle de décision. Pour illustrer les $\boldsymbol{PDM}$, on prend souvent des exemples issus de la robotique mobile (avec les positions pour états, les commandes comme actions, les mouvements comme transitions et l'accomplissement / échec de tâches comme gains / pénalités).

1.2. Processus stochastique

Les problèmes de prise de décisions dans l'incertain sont des **problèmes** dont la dynamique est incertaine appelés aussi **problèmes d'optimisation stochastiques [22]**. On peut définir un processus stochastique $\{\mathbf{X_t}\}_{\mathbf{t\in T}}$ comme étant une collection de variables aléatoires définit sur un même espace probabilisé, la variable $\mathbf{X_t}$ représente l'état du processus au temps t et l'ensemble de toutes les valeurs possibles pour cette variable est appelé l'espace des états du processus est sera noté **E.** Ces différentes variables aléatoires $\mathbf{X_t}, \boldsymbol{t} = \mathbf{0}, \mathbf{1}\ldots$ ne sont en général pas indépendantes les unes des autres. Ainsi, et Pour spécifier entièrement un processus stochastique, il suffit de spécifier :

1. La loi de probabilité de la première variable aléatoire $\mathbf{X_0}$, qui spécifie donc l'état du processus lors de la première observation.
2. Pour toute valeur de $\boldsymbol{t} > \mathbf{1}$ la probabilité conditionnelle :

$$\mathbb{P}(\mathbf{X_t} = \boldsymbol{j} | \mathbf{X_0} = \mathbf{i_0}, \mathbf{X_1} = \mathbf{i_1}, \ldots, \mathbf{X_{t-1}} = \mathbf{i_{t-1}})$$

1.3. Chaine de Markov

Soit $(\mathbf{X_n})_{\boldsymbol{n}\in\mathbb{N}}$ une suite de variable aléatoire ($\mathbf{v.a}$) à valeurs dans un espace **E** fini ou dénombrable, appelé espace d'états. On dit que $(\mathbf{X_n})_{\boldsymbol{n}\in\mathbb{N}}$ est une chaîne de Markov si :

$\mathbb{P}(\mathbf{X_{n+1}} = \boldsymbol{j} | \mathbf{X_n} = \mathbf{i_n}, \mathbf{X_{n-1}} = \mathbf{i_{n-1}}, \ldots, \mathbf{X_1} = \mathbf{i_1}, \mathbf{X_0} = \mathbf{i_0}) = \mathbb{P}(\mathbf{X_{n+1}} = \boldsymbol{j} | \mathbf{X_n} = \mathbf{i_n})$, pour tout entier $\mathbf{n} \in \mathbb{N}$, pour tout état **j** et pour toute suite d'états $\mathbf{i_0}, \mathbf{i_1}, \ldots, \mathbf{i_{n-1}}, \mathbf{i_n}$.

Si de plus la quantité $\mathbb{P}(\mathbf{X_{n+1}} = \boldsymbol{j} | \mathbf{X_n} = \mathbf{i_n})$ ne dépend pas de n, alors la chaîne de Markov $(\mathbf{X_n})_{\boldsymbol{n}\in\mathbb{N}}$ est dite homogène.

On peut voir une chaîne de Markov $(\mathbf{X_n})_{\boldsymbol{n}\in\mathbb{N}}$ comme une promenade dans l'espace d'états **E**, la variable ($\mathbf{X_n}$ indiquant l'état dans lequel on est à l'instant **n**. La $\mathbf{v.a.}$ $\mathbf{X_0}$ Représente l'état initial duquel démarre la chaîne. Selon le contexte, $\mathbf{X_0}$ pourra être aléatoire ou déterministe. La propriété de Markov signifie que, connaissant le dernier état visité (disons

à l'instant $\mathbf{n}$.), la loi du prochain état visité (la loi de$(\mathbf{X_{n+1}})_{n\in\mathbb{N}}$) ne dépend pas des états visités depuis l'instant $\mathbf{0}$ jusqu'à l'instant $\mathbf{n-1}$. Plus prosaïquement, on dit que : *Conditionnellement au présent, le futur ne dépend pas du passé*

1.4. PDM

Les processus décisionnels de Markov sont définis comme des processus stochastiques contrôlés satisfaisant la propriété de Markov, assignant des récompenses aux transitions d'états. Ces processus permettent de modéliser de nombreux phénomènes, ils sont par exemples utilisés pour des problèmes de télécommunication, de théorie du signal, d'imagerie informatique, etc.

Un décideur, un agent ou un contrôleur (auquel nous faisons référence), est confronté au problème, voire à l'opportunité, d'influencer le comportement d'un système probabiliste à mesure qu'il évolue dans le temps. Il le fait en prenant des décisions ou en choisissant des actions. Son but est de choisir une séquence d'actions qui permettent au système de fonctionner de manière optimale en respectant un critère de performance prédéterminé. Vu que le système, lequel nous modélisons est en cours, son état avant la décision de demain dépend de la décision d'aujourd'hui. Par conséquent, les décisions ne doivent pas être prises de manière évidente, mais doivent anticiper les opportunités et les coûts (ou les récompenses) associés aux futurs états du système. Les PDM offrent un cadre théorique simple et un ensemble d'algorithmes, permettant le bon choix de la décision

1.4.1. Cadre théorique

Un **PDM** est définit par un tuplet : (**S**, **A**, **T**, **P**, **R**) où :

- **S** est l'espace d'états dans lequel évolue le processus
- **A** est l'espace des actions qui contrôlent la dynamique de l'état
- **T** est l'espace des temps
- $\mathbf{P}(.)$ Sont les probabilités de transition entre états
- $\mathbf{R}(.)$ Est la fonction de récompense sur les transitions entre états.

Le domaine **T** des étapes de décision est un ensemble discret, assimilé à un sous-ensemble de $\mathbb{N}$, qui peut être fini ou infini (on parle d'horizon fini ou d'horizon infini). Les domaines **S** et **A** sont supposés finis et constants tout au long du processus. Les probabilités de transition caractérisent la dynamique de l'état du système. Pour une action **a** fixée, $\mathbf{P}(\mathbf{s}'|\mathbf{s},\mathbf{a})$ représente la probabilité que le système passe dans l'état $\mathbf{s}'$ après avoir exécuté l'action **a** dans l'état **s**. Les distributions $\mathbf{P}()$ vérifient la propriété fondamentale qui donne son nom aux processus décisionnels de Markov considérés ici.

$$\mathbf{P}(\mathbf{s_{t+1}}|\mathbf{s_0},\mathbf{a_0},\mathbf{s_1},\mathbf{a_1},\dots,\mathbf{s_t},\mathbf{a_t}) = \mathbf{P}(\mathbf{s_{t+1}}|\mathbf{s_t},\mathbf{a_t}) \quad \textbf{(1)}$$

Comme résultat d'avoir choisi l'action **a** dans l'état **s** à l'instant **t**, l'agent décideur reçoit une récompense, ou revenu, $\mathbf{R_t} = \mathbf{R}(\mathbf{s},\mathbf{a}) \in \mathbb{R}$. Les valeurs positives de $\mathbf{R_t}$ peuvent être considérées comme des gains, ainsi que les valeurs négatives comme des coûts. Cette récompense peut être instantanément perçue à la date **t**, ou accumulée de la date **t** à la date **t+1**, elle dépend de l'état et de l'action choisie à l'instant courant et dans certains cas elle peut dépendre aussi de l'état suivant. Par ailleurs, comme pour **S** et **A**, les fonctions de transition et de récompense peuvent elles-mêmes varier au cours du temps, auquel cas on les note $\mathbf{P_t}()$

et $\mathbf{R_t}()$. Lorsque ces fonctions ne varient pas, on parle de processus stationnaires : pour tous $\mathbf{t} \in \mathbf{T}$, $\mathbf{P_t}() = \mathbf{p}()$ et $\mathbf{R_t}() = \mathbf{R}()$. Par la suite, nous supposerons vérifiée cette hypothèse de stationnarité dans l'étude des **PDM** à horizon infini.

1.4.2. Les politiques

On nomme politique (notée $\mathbf{\Pi}$), ou stratégie, la procédure suivie par l'agent pour choisir à chaque instant l'action à exécuter. Une politique peut être déterministe ou aléatoire et peut se baser sur l'historique $\mathbf{H_t}$ du processus, où ne considérer que l'état courant $\boldsymbol{s_t}$ et, est dite stationnaire s'elle ne dépend pas du temps, pour cela on distingue différents types de politique :

- **Politique stochastique :** Une politique stochastique est une famille de distributions de probabilité selon laquelle une action $\boldsymbol{a}$ doit être sélectionnée pour chaque historique observé $\boldsymbol{h^t}$**.** Elle appartienne à l'ensemble de toute les stratégies stochastiques : $\mathbf{\Pi}^{\mathbf{S}}$ et on la note $\boldsymbol{\pi} = \{\boldsymbol{\pi}^{\mathbf{t}}(\mathbf{a}|\mathbf{h}^{\mathbf{t}}), \forall \mathbf{t}, \forall \mathbf{h}^{\mathbf{t}}, \forall \mathbf{a}\}$. $\boldsymbol{\pi}^{\mathbf{t}}(\mathbf{a}|\mathbf{h}^{\mathbf{t}})$ Est la probabilité de choisir l'action $\mathbf{a}$ au pas de temps $\mathbf{t}$ si on a observé l'historique $\mathbf{h}^{\mathbf{t}}$**.**
- **Politique déterministe :** Une politique dans l'ensemble $\mathbf{\Pi}^{\mathbf{D}}$, qui définit précisément l'action qui doit être sélectionnée en se basant sur l'historique $\mathbf{h}^{\mathbf{t}}$**.** On la note $\boldsymbol{\pi} = \{\boldsymbol{\pi}^{\mathbf{t}}(\mathbf{h}^{\mathbf{t}}), \forall \mathbf{t}, \forall \mathbf{h}^{\mathbf{t}}\}$. $\boldsymbol{\pi}^{\mathbf{t}}(\mathbf{h}^{\mathbf{t}}) \in \mathbf{A}$ Représente l'action qui doit être sélectionnée au pas de temps $\boldsymbol{t}$ si on a observé l'historique $\mathbf{h}^{\mathbf{t}}$**.**
- **Politique Markovienne :** On parle de politique (stochastique ou déterministe) markovienne $\boldsymbol{\pi} \in \mathbf{\Pi}^{\mathbf{MSD}}$,lorsque la politique ne dépend que de l'état courant $\mathbf{s}^{\mathbf{t}}$ et non de l'ensemble de l'historique $\mathbf{h}^{\mathbf{t}}$**.** Dans le cas contraire, on parle de politique histoire-dépendante.
- **Politique (markovienne) stationnaire :** On appelle politique (stochastique ou déterministe) stationnaire $\boldsymbol{\pi} \in \mathbf{\Pi}^{\mathbf{MS}}$ **,** toute politique markovienne qui ne dépend pas du temps ($\forall(\mathbf{t}, \mathbf{t}') \in \mathbf{T}^2), \boldsymbol{\pi}^{\mathbf{t}} = \boldsymbol{\pi}^{\mathbf{t}'}$).

Nous noterons $\mathbf{\Pi}$ l'ensemble le plus général des politique (histoire-dépendantes stochastiques), $\mathbf{\Pi}^{\mathbf{MS}}$ l'ensemble des politiques markovienne stationnaires, $\mathbf{\Pi}^{\mathbf{MSD}}$ l'ensemble des politiques markoviennes stationnaires déterministes $\mathbf{\Pi}^{\mathbf{MSD}} \subset \mathbf{\Pi}^{\mathbf{MS}} \subset \mathbf{\Pi}$ et $\mathbf{\Pi}^{\mathbf{MSD}} \subset \mathbf{\Pi}^{\mathbf{D}} \subset \mathbf{\Pi}$ **.** Notons que pour une politique $\boldsymbol{\pi} \in \mathbf{\Pi}^{\mathbf{MS}}$donnée, l'état du système suit une chaîne de Markov de probabilités de transition.

1.4.3. Critère d'optimisation

Résoudre un **PDM** c'est chercher parmi une famille de politiques celles qui optimisent un critère de performance donné pour le **PDM** considéré. Ce critère a pour ambition de caractériser les politiques qui permettent de générer des séquences de récompenses les plus importantes possibles le long d'une trajectoire. Suivant les caractéristiques du problème, on utilise un critère d'optimalité approprié. En termes formels, cela revient toujours à évaluer une politique sur la base d'une mesure du cumul espéré des récompenses instantanées le long d'une trajectoire, comme on peut le voir sur les critères les plus étudiés au sein de la théorie des **PDM**, qui sont respectivement :

- **Critère fini** : $\mathbf{E}[\mathbf{R_0} + \mathbf{R_1} + \cdots + \mathbf{R_{N-1}} | \mathbf{S_0}]$

- **Critère γ-pondéré** : $\mathbf{E[R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots + \gamma^t R_t + \cdots | S_0]}$
- **Critère total** : $\mathbf{E[R_0 + R_1 + \cdots + R_t + \cdots | S_0]}$

Les deux caractéristiques communes à ces quatre critères sont en effet d'une part leur formule additive en $\mathbf{R_t}$, qui est une manière simple de résumer l'ensemble des récompenses reçues le long d'une trajectoire et, d'autre part, l'espérance **E[.]** qui est retenue pour résumer la distribution des récompenses pouvant être reçues le long des trajectoires, pour une même politique et un même état de départ.

1.4.4. Fonction de valeurs

Ces critères permettent de définir une fonction de valeur $\boldsymbol{V^{\pi}}$: $\boldsymbol{S} \rightarrow \mathbb{R}$ qui, pour une politique $\boldsymbol{\pi}$ fixée, associe à tout état initial $\mathbf{s} \in \mathbf{S}$ la valeur du critère considéré en suivant $\boldsymbol{\pi}$ à partir de **s**.

Le critère fini suppose que l'agent doit contrôler le système en **N** étapes. Ce critère conduit naturellement à définir la fonction de valeur qui associe à tout état **s** l'espérance de la somme des **N** prochaines récompenses en suivant la politique $\boldsymbol{\pi}$ à partir de **s** :

$$\forall \mathbf{s} \in \mathbf{S} \;\; \mathbf{V_N^{\pi}(s) = E^{\pi}\left[\sum_{t=0}^{N-1} R_t | s_0 = s\right]}. \qquad \textbf{(1.1)}$$

Le critère γ-pondéré, ou critère actualisé, est le critère à horizon infini le plus classique. La fonction de valeur du critère **γ-pondéré** est celle qui associe à tout état **s**, la limite lorsque **N** tend vers l'infini de l'espérance en suivant la politique **Π** à partir de **s** de la somme des **N** futures prochaines récompenses, pondérées par un facteur d'actualisation :

$$\forall \mathbf{s} \in \mathbf{S} \;\; \mathbf{V_{\gamma}^{\pi}(s) = E^{\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_t | s_0 = s\right]}. \qquad \textbf{(1.2)}$$

Le critère total utilisé dans certains cas particuliers à horizon infini (**γ=1**) a pour fonction de valeur :

$$\forall \mathbf{s} \in \mathbf{S} \;\; \mathbf{V^{\pi}(s) = E^{\pi}\left[\sum_{t=0}^{\infty} R_t | s_0 = s\right]}. \qquad \textbf{(1.3)}$$

1.4.5. Equations d'optimalité

A chaque critère correspond une équation d'optimalité qui détermine les fonctions de valeurs optimales ainsi que les politiques optimales du problème de décision markovien à résoudre.

On se limite dans ce travail au **Critère γ-pondéré** et que le **PDM** considéré est stationnaire. Soit $\boldsymbol{\pi}$ une politique markovienne stationnaire. Nous utilisons la notation vectorielle qui vient par la suite dans ce chapitre, pour $\boldsymbol{v} \in \mathbf{V}$, on définit l'opérateur $\mathbf{L_{\pi}}$ de $\boldsymbol{\mathcal{V}}$ dans $\boldsymbol{\mathcal{V}}$, espace vectoriel muni de la norme

max : $\forall \mathbf{V} \in \boldsymbol{\mathcal{V}}, \|\mathbf{V}\| = \max_{s \in S} |\mathbf{V(s)}|$, par :

$$\forall\, \mathbf{V} \in \mathcal{V}, \mathbf{L_\pi V} = \mathbf{R_\pi} + \boldsymbol{\gamma}\mathbf{P_\pi V} \tag{1.4}$$

Où $\mathbf{R_\pi}$ ($\mathbf{P_\pi}$) est le vecteur récompense (matrice de transition) associé à la politique $\boldsymbol{\pi}$.

Théorème 1 : Soient $\mathbf{N} < \infty$ et $\boldsymbol{\pi}$ une politique markovienne déterministe. Alors la fonction de valeur optimale V_π^* est solution du système d'équations linéaires suivant :

$$\mathbf{V_{n+1}(s)} = \max_{\mathbf{a \in A(s)}} \left(\mathbf{R(s,a)} + \sum_{\mathbf{s' \in S}} \mathbf{P(s'|s,a)V_n(s')} \right) \tag{1.5}$$

Pour $\mathbf{n = 0, 1, \ldots, N-1}$ et $\mathbf{V_0 = 0}$.
La politique optimale est ainsi donnée par l'équation ci-dessous.

$$\boldsymbol{\pi}^*\mathbf{(s)} = \underset{\mathbf{a \in A(s)}}{\mathbf{argmax}} \left(\mathbf{R(s,a)} + \sum_{\mathbf{s' \in S}} \mathbf{P(s'|s,a)V_n(s')} \right) \tag{1.6}$$

Théorème 2 : Soient $\boldsymbol{\gamma} < \mathbf{1}$ et $\boldsymbol{\pi}$ une politique stationnaire markovienne. Alors $\mathbf{V_\gamma^\pi}$ est l'unique solution de l'équation $\mathbf{V = L_\pi V}$.

$$\forall \mathbf{s} \in \mathbf{S} \qquad \mathbf{V(s)} = \mathbf{R_\pi(s)} + \boldsymbol{\gamma} \sum_{\mathbf{s' \in S}} \mathbf{P_\pi(s'|s)V(s')} \tag{1.7}$$

La résolution d'un PDM revient à résoudre le problème d'optimisation $\mathbf{V^*(s)} = \max_{\boldsymbol{\pi}} V_\gamma^{\boldsymbol{\pi}}$. On définit maintenant l'opérateur $\mathbf{L}$ de l'ensemble des fonctions de valeur $\mathcal{V}$ dans lui-même, nommé opérateur de programmation dynamique par :

$$\forall\, \mathbf{V} \in \mathcal{V} \text{ et } \forall\, \mathbf{s} \in \mathbf{S} \qquad \mathbf{LV(s)} = \max_{\mathbf{a \in A(s)}} \left\{ \mathbf{R(s,a)} + \boldsymbol{\gamma} \sum_{\mathbf{s' \in S}} \mathbf{p(s'|s,a)V(s')} \right\} \tag{1.8}$$

Soit encore en notation vectorielle par :

$$\forall\, \mathbf{V} \in \mathcal{V} \qquad \mathbf{LV} = \max_{\boldsymbol{\pi}} \{\mathbf{R_\pi} + \boldsymbol{\gamma}\mathbf{P_\pi V}\} \tag{1.9}$$

Théorème 3 (Equation de Bellman**) :** Soient $\boldsymbol{\gamma} < \mathbf{1}$. Alors V_γ^* est l'unique solution de l'équation $\boldsymbol{L}\mathbf{V} = \mathbf{V}$.

$$\forall\, \mathbf{s} \in \mathbf{S} \qquad \boldsymbol{V}\mathbf{(s)} = \max_{\boldsymbol{a \in A(s)}} \left\{ \mathbf{R(s,a)} + \boldsymbol{\gamma} \sum_{\mathbf{s' \in S}} \mathbf{p(s'|s,a)V(s')} \right\} \tag{1.10}$$

Théorème 4 : Soit $\boldsymbol{\gamma} < \mathbf{1}$. L'opérateur de programmation dynamique $\mathbf{L}$ défini par, $\mathbf{LV} = \max_{\boldsymbol{\pi}} \{\mathbf{R_\pi} + \boldsymbol{\gamma}\mathbf{P_\pi V}\}$ est une contraction sur $\mathcal{V}$. Ce qui implique que :

1) Il existe un unique point fixe $\mathbf{V_\gamma^*} \in \mathcal{V}$ tel que $\mathbf{V_\gamma^*} = \mathbf{V_\gamma^*}$.
2) Pour tout $\mathbf{V_0} \in \mathcal{V}$, la suite $(\mathbf{V_n})_{\mathbf{n>0}}$ définie par $\mathbf{V_{n+1}} = \mathbf{LV_n} = \mathbf{L^{n+1}V_0}$ converge vers $\mathbf{V_\gamma^*}$.

Théorème 5 (Politique optimale) **:** Soit $\boldsymbol{\gamma} < \mathbf{1}$ **et** $\mathbf{V_\gamma^*}$ solution de l'équation de Bellman**.**
Alors toute politique stationnaire $\boldsymbol{\pi}^*$ définie par :

$$\forall\, \mathbf{s} \in \mathbf{S} \qquad \boldsymbol{\pi}\mathbf{(s)} \in \mathbf{argmax_{a \in A}} \left\{ \mathbf{R(s,a)} + \boldsymbol{\gamma} \sum_{\mathbf{s' \in S}} \mathbf{p(s'|s,a)}\boldsymbol{V_\gamma^*}\mathbf{(s')} \right\} \tag{1.11}$$

Est une politique optimale.

Dans toute la suite, nous allons considérer les suppositions suivantes :

- **Supposition 1.1**
  Les récompenses stationnaires, ainsi que la probabilité de transition ; $\mathbf{r(s,a)}$ et $\mathbf{p(j|i,a)}$ ne varient pas au court du temps.
- **Supposition 1.2**
  $|\mathbf{r(s,a)}| \leq \mathbf{M} \leq \infty$ Pour tout$\mathbf{a} \in \mathbf{A_s}$ et $\mathbf{s} \in \mathbf{S}$.
- **Supposition 1.3**
  Les récompenses futures, sont supposées actualisées et sont liées à un facteur d'actualisation $\boldsymbol{\alpha}$, avec $0 \leq \boldsymbol{\alpha} < \mathbf{1}$.

1.5. Applications

L'étude faite en **[7]**, a fait explorer l'étendue du champ applicatif des PDM au travers de deux problématiques. La première, de nature industrielle, propose l'étude numérique de l'optimisation d'un processus d'intégration lanceur en collaboration avec Airbus DS. Il s'agit d'un cas particulier des problèmes de gestion d'inventaire dans lequel un calendrier de tirs joue un rôle central. La modélisation adoptée entraîne l'impossibilité d'appliquer les procédures d'optimisation classiques liées au formalisme des processus markoviens décisionnels. Les algorithmes qui ont utilisé alors pour ces problèmes sont basés sur des simulations qui rendent des stratégies optimales non triviales et qui sont utilisables dans la pratique. La prévision de l'évolution du risque peut être un élément précieux pour la prise de décision ; elle permet par exemple d'évaluer à quel moment et où il faut intervenir lorsque les ressources sont limitées. Ce sera le cas pour le management des urgences lors d'un désastre humanitaire, **Hélène** et **Juliette**, dans **[21]**, on fait une démarche usuellement adoptée, appelée **QRA** (Quantitative Risk Assessment), comporte plusieurs phases comme « l'évaluation du risque « . Afin de présenter une approche permettant de prédire l'évolution du risque dans un futur à moyen et long terme, offrant ainsi au décideur une estimation du temps dont il dispose pour intervenir et pour planifier ses actions. **Olivier Sigaud [13]** a présenté des nombreuses applications du **PDM** dans le domaine de l'intelligence artificielle, comme aux jeux stochastiques, Apprentissage en ligne de la manipulation de micro-objets, Robotique mobile, Planification d'opérations et d'autres applications. Ainsi, on peut rencontrer beaucoup de systèmes qui peuvent se modéliser avec des chaînes de Markov : en Sûreté de Fonctionnement (**Corazza**, 1975) en **[19]**, (**Schoenig** *et al.*, 2003a) et (**Schoenig** *et al.*, 2003b) en **[9]** et **[18]** , mais aussi finance, dans le domaine de la biologie dans **[12]**, en socio-politique dans **[20]**, en imagerie spatiale dans **[6]**, et surtout une bonne étude de **Rabiner** était fait en 1989 dans le domaine du traitement de la parole en **[16]**. Christophe a fait une étude détaillée en **[8]** au jeu de Tetris, avec une application sur laquelle se sont penchés plusieurs travaux de la littérature. **Bäuerle, Nicole** dans son livre **[4],** se concentrait sur les applications financières et les domaines dans lesquels les PDM sont utilisés pour résoudre des nombreux problèmes, par exemple dans la planification de la production, le contrôle des stocks, la gestion des opérations, l'ingénierie, la biologie et les statistiques.

1.6. Conclusion

Le cadre des processus décisionnels de Markov, avec les critères d'optimalité et les algorithmes de résolution que nous avons présentés dans ce chapitre, constitue un outil méthodologique de base pour les problèmes de décision séquentielle dans l'incertain et devient de plus en plus utilisé en intelligence artificielle. Toutefois, ce cadre théorique est limité par le fait qu'il suppose de la part de l'agent une parfaite connaissance de son état à chaque instant, des fonctions de transition et de récompense qui définissent le problème auquel il est confronté. Ces algorithmes ainsi présentés ne peuvent appliquées aux problèmes où l'agent ne connait pas son état avec exactitude ou ne connait pas les lois de probabilités de

transition et de récompenses. Nous aborderons au chapitre 2 comment formaliser un PDM où l'état de l'agent n'est pas connu avec exactitude mais selon une distribution de probabilité qui définit une observation partielle de son état. Nous expliquerons aussi comment l'apprentissage par renforcement permet résoudre des problèmes où l'agent ne connait pas à priori les fonctions de transition qui définissent le dynamique du système ainsi que les récompenses qui évaluent ces performances.

Chapitre II : Algorithmes de résolution PDM sous le critère d'actualisation

2.1. Introduction

Dans ce chapitre, nous allons étudier le processus de décision markoviens à horizon infini avec le critère d'actualisation. Ces modèles sont les mieux compris de tous les critères des PDM. La résolution de ces modèles fournit, une base pour la théorie des modèles avec d'autres critères d'optimalité. Les résultats pour les modèles à prix réduits sont remarquables en ce qu'ils sont valables quelle que soit la structure des chaînes de Markov générée par les politiques stratégiques.

Nous nous concentrons sur l'équation d'optimalité et nous utilisons la théorie des questions sur les espaces linéaires normés pour assurer l'existence de sa solution et la convergence des algorithmes de résolution. Ainsi que les propriétés des solutions de l'équation d'optimalité et nous fournissons des conditions garantissant la sortie des stratégies optimales. Tout au long de ce chapitre, nous accordons une attention particulière aux algorithmes suivants, permettant de rechercher les stratégies optimales :

- **Programmation linéaire**
- **Itération de la valeur et ses variantes**
- **Itération de politique et ses variantes**

Durant cette étude, nous étudions les PDM à horizon infini avec le critère d'optimalité de la récompense totale attendue. Ces modèles sont les mieux compris de tous les problèmes de décision de Markov à horizon infini. Les résultats de ces modèles fournissent un Standard pour la théorie des modèles avec d'autres critères d'optimalité. Les résultats des modèles ignorés sont importants dans la mesure où ils sont indépendants de la chaîne de Markov générée par les stratégies stationnaires. Dans la suite, nous décrivons les méthodes classiques de résolution exactes des **PDM** à horizon fini et infini. Nous décrivons aussi l'approche plus récente de planification par inférence qui voit la résolution d'un **PDM** comme un problème d'inférence dans un modèle graphique.

2.2. Algorithmes de résolution

Algorithme 1 : Algorithme d'induction rétroactif pour le critère fini

Entrés : $\mathbf{E, A, P, R}$
Sorties : $\mathbf{V^*, \pi^*}$
1 $\mathbf{V^*(s) \leftarrow R(s), \forall s \in E;}$
2 Pour n $\mathbf{\leftarrow N-1}$ **Jusqu'à 0 Faire**
3 **Pour tout s** $\mathbf{\in E}$ **Faire**
4 $\mathbf{V_n^*(s) = \max_{a\in A}\{ R(s,a) + \sum_{S'\in E} P(s',a,s)\, V_{n-1}^*(s')\}}$
5 $\mathbf{\pi_n^* \in \mathrm{Argmax}_{a\in A}\{ R(s,a) + \sum_{S'\in E} P(s',a,s)\, V_{n-1}^*(s')\}}$
6 **Finpour**
7 finpour

Figure 1 : Algorithme d'induction rétroactif pour le critère fini.

1.5.1 Le critère fini

Pour ce critère d'horizon fini. Les équations d'optimalité **(1.5)** et **(1.6)** permettent de calculer récursivement à partir de la dernière étape les fonctions de valeur optimales selon l'algorithme **1.2** d'induction rétroactif suivant.

Algorithme 2 : Algorithme d'induction rétroactif pour le critère fini

1: Pour tout s $\in$ **S Faire** $\mathbf{V_0(s) = 0}$

2: Pour n←0 **jusqu'à** N-1 **Faire**

3: Pour tout s $\in$ **S Faire**

$$\mathbf{V}_{n+1}(\mathbf{s}) \leftarrow \max_{a \in A(s)} \left(R(s,a) + \sum_{s' \in S} P(s'|s,a)\mathbf{V}_n(s') \right)$$

4: Pour tout $s \in S$ **faire**

$$\boldsymbol{\pi}^*(\mathbf{s}) \leftarrow \underset{a \in A(s)}{\mathbf{argmax}} \left(R(s,a) + \sum_{s' \in S} P(s'|s,a)\mathbf{V}_n(s') \right)$$

5: Retourner V, $\boldsymbol{\pi}$

Figure 2 : Algorithme d'induction rétroactif pour le critère fini

Il est clair que La complexité temporelle et spatiale de cet algorithme est en $\mathcal{O}(\mathbf{N|S|^2|A|})$, ce qui le rend couteux pour des espaces d'états et d'actions assez grands.

1.5.2. Le critère γ-pondéré

Il existe trois grandes familles d'algorithmes de résolution des **PDM** sous le critère **γ-pondéré** : la programmation linéaire, l'itération sur les valeurs et l'itération sur les politiques.

a. Algorithme de programmation linéaire

On montre dans la preuve du théorème 1.3 que si $\mathbf{V} \in \mathcal{V}$ minimise la fonction $\sum_{s \in S} \mathbf{V}(s)$ sous la contrainte $\geq \mathbf{LV}$, alors $\mathbf{V} = \mathbf{V}_\gamma^*$. C'est ainsi que l'algorithme de programmation linéaire proposée initialement par [D'] se présente comme suit.

LITTMAN **[23]** a montré que si n et m sont les tailles respectives de S et A, avec **P**() et **R**() codées sur b bits, la complexité d'un tel algorithme de programmation linéaire sur les rationnels est polynomiale en **|S|**, **|A|**, b, avec des temps de résolution assez lents.

Algorithme 3 : Programmation linéaire, critère γ-pondéré

1: **Résoudre**: $\underset{V \in \mathcal{V}}{\text{Min}} \sum_{s \in S} V(s)$

Avec: $\mathbf{V(s) \geq R(s,a) + \gamma \sum_{s' \in S} p(s' \backslash s,a) V_n(s')}, \forall\, \mathbf{s \in S}, \forall\, \mathbf{a \in A(s)}$

2 : Pour tout $\boldsymbol{s \in S}$ **faire**

$$\boldsymbol{\pi(s) \in \text{argmax}_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} p(s' \backslash s,a) V_n(s') \right\}}$$

3 : Retourner V, $\boldsymbol{\pi}$

Figure 3 : Programmation linéaire, critère γ-pondéré

b. Algorithme d'itération sur les valeurs

L'approche la plus classique se base aussi sur la résolution directe de l'équation d'optimalité de Bellman $\mathbf{V = LV}$, en utilisant pour cela une méthode itérative de type point fixe, d'où son nom anglais de « value iteration » **[14]**. A partir d'une valeur initiale arbitraire de chaque état, l'algorithme améliore itérativement l'évaluation des états par approximations successives en utilisant l'équation de Bellman **(1.9)**. La valeur d'un état à l'itération **n** est calculée à partir de sa valeur à l'itération **n-1**. Le processus itératif s'arrête lorsque la différence entre les valeurs successives de tous les états est inférieure à un paramètre $\boldsymbol{\varepsilon}$.

Algorithme 4 : Itération de la valeur, critère γ-pondéré

1: Initialiser $\mathbf{V_0} \in \boldsymbol{\mathcal{V}}$

2: $\mathbf{n \leftarrow 0}$

3 : Répéter

4 : Pour tout $\mathbf{s \in S}$ **faire**

$$\mathbf{V_{n+1}(s) \leftarrow \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s'} p(s' \backslash s,a) V_n(s') \right\}}$$

5: $\mathbf{n \leftarrow n + 1}$

6 : Jusqu'à $\mathbf{\|V_{n+1} - V_n\| < \varepsilon}$

7 : Pour tout $\boldsymbol{s \in S}$ **faire**

$$\boldsymbol{\pi(s) \in \text{argmax}_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in S} p(s' \backslash s,a) V_n(s') \right\}}$$

8: Retourner $\mathbf{V_n}$, $\boldsymbol{\pi}$

Figure 4 : Algorithme d'itération de la valeur, critère γ-pondéré

Il est alors établi qu'un nombre maximum d'itérations polynomial en **|S|, |A|**, b, **1/(1− γ) log(1/(1− γ))** est nécessaire pour atteindre $\boldsymbol{\pi}^*$, chaque itération étant de complexité **O(|A||S|²)**. Au-delà de ce nombre d'itérations, la suite $\mathbf{V}_{\boldsymbol{n}}$ est de plus en plus proche de $\mathbf{V}^*$ mais la politique correspondante $\boldsymbol{\pi}_{\boldsymbol{n}}$ ne change plus.

Il est possible d'améliorer la vitesse de convergence de l'algorithme d'itération sur les valeurs en modifiant légèrement le calcul de $\mathbf{V}_{\boldsymbol{n+1}}$. L'idée consiste à utiliser $\mathbf{V}_{\boldsymbol{n+1}}(\boldsymbol{s})$ à la place de $\mathbf{V}_{\boldsymbol{n}}(\boldsymbol{s})$ lorsque cette valeur a déjà été calculée. On définit ainsi l'algorithme de Gauss-Seidel (algorithme 1.5).

Algorithme 5 : Itération de la valeur, critère γ-pondéré, Gauss-Seidel

1 : Initialiser $\mathbf{V_0} \in \mathcal{V}$

2 : $Convergence \leftarrow Faux$

3 : Tant que ($Convergence = Faux$) **Faire**

4 : $Convergence \leftarrow Vrai$

5 : **Pour** tout $\boldsymbol{s} \in \boldsymbol{S}$ **faire**

a. $\mathbf{V_{tmp}} = \mathbf{max_{a\in A}}\{\mathbf{r(s,a)} + \boldsymbol{\gamma} \sum_{s'} \mathbf{p(s'|s,a)V(s')}\}$

b. **Si** $\left(\left\|\mathbf{V_{tmp}} - \mathbf{V(s)}\right\| \geq \boldsymbol{\varepsilon} \right)$**Alors Quitte = faux**

c. $\mathbf{V(s)} = \mathbf{V_{tmp}}$

6 : Pour tout $\boldsymbol{s} \in \boldsymbol{S}$ **faire**

$$\boldsymbol{\pi}(\mathbf{s}) \in \mathbf{argmax_{a\in A}}\left\{\mathbf{r(s,a)} + \boldsymbol{\gamma} \sum_{\mathbf{s'\in S}} \mathbf{p(s'|s,a)V(s')}\right\}$$

7 : Retourner $\boldsymbol{V_n}, \boldsymbol{\pi}$

Figure 5 : Algorithme d'itération de la valeur, critère γ-pondéré, Gauss-Seidel

c. Algorithme d'itération sur les politiques

L'algorithme d'itération sur les politiques (Algorithme 1.6**)** est aussi un algorithme itératif utilisant le principe d'optimalité de Bellman afin de résoudre des **PDM** à horizon fini ou infini en construisant directement la politique optimale.

Algorithme 6 : Itération de la politique, critère γ-pondéré

1 : Initialiser $\boldsymbol{\pi_0} \in \mathbf{D}$

2 : $\mathbf{n} \leftarrow \mathbf{0}$

3 : Répéter

4 : Résoudre le système :

$$\forall \mathbf{s} \in \mathbf{S}, \quad \mathbf{V_n(s) = r\big(s, \pi_n(s)\big) + \gamma \sum_{s' \in S} p(s'|s, \pi_n(s)) V_n(s')}$$

5 : **Pour** tout $\boldsymbol{s} \in \boldsymbol{S}$ **faire**

$$\boldsymbol{\pi_{n+1}(s) \in \mathrm{argmax}_{a \in A} \left\{ r(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V_n(s') \right\}}$$

6 : $\mathbf{n} \leftarrow \boldsymbol{n} + \mathbf{1}$

7 : jusqu'à $\boldsymbol{\pi_{n+1} = \pi_n}$

8 : Retourner $\mathbf{V_n}, \boldsymbol{\pi_{n+1}}$

Figure 6 : Algorithme d'itération de la politique, critère γ-pondéré

L'algorithme part d'une politique initiale arbitraire $\boldsymbol{\pi_0}$. A chaque itération, la politique courante est évaluée puis améliorée. L'évaluation utilise l'équation de Bellman et revient à résoudre un système d'équations linéaires. La phase d'amélioration consiste à mettre à jour la politique courante en utilisant les valeurs calculées lors de la phase d'évaluation.

L'algorithme converge lorsqu'aucune modification de la politique n'est plus possible.

La complexité en temps de cet algorithme dépend de la complexité d'une itération et du nombre d'itérations. La complexité de la phase d'évaluation d'une politique est en $\mathbf{O(|S|^3)}$ et celle de l'amélioration est en $\mathbf{O(|A||S|^2)}$**.**

Le nombre d'itérations de l'algorithme dépend fortement de la politique initiale $\boldsymbol{\pi_0}$. Il existe $|\mathbf{A}|^{|S|}$ politiques possibles. Étant donné qu'à chaque itération la politique mise à jour domine strictement la politique précédente, le nombre d'itérations sera, au pire cas, exponentiel en nombre d'états.

Il est possible d'améliorer l'efficacité de cet algorithme en simplifiant la phase d'évaluation de la politique courante $\boldsymbol{\pi_n}$ en résolvant l'équation $\mathbf{V_n = L_{\pi_n} V_n}$ de manière itérative, comme pour

l'itération sur les valeurs. L'utilisation de ce principe conduit à l'algorithme modifié d'itération sur les politiques (algorithme 1.6).

Algorithme 7 : Itération de la politique modifié, critère γ-pondéré

1: Initialiser $\boldsymbol{\pi_0} \in \mathbf{D}$

2: **Répéter**

3: Initialiser $\mathbf{V_0} \in \boldsymbol{\mathcal{V}}$

4 : $\mathbf{n} \leftarrow \mathbf{0}$

5 : $Convergence \leftarrow Faux$

6 : **Tant que** $(Convergence = Faux)$ **Faire**

7 : $Convergence \leftarrow Vrai$

8 : **Pour** tout $\boldsymbol{s} \in \boldsymbol{S}$ **faire**

$$\mathbf{V_{tmp}} = \mathbf{r}(\mathbf{s}, \boldsymbol{\pi_n}(\boldsymbol{s})) + \boldsymbol{\gamma} \sum_{\mathbf{s'}} \mathbf{p}(\mathbf{s'}|\mathbf{s}, \boldsymbol{\pi_n}(\boldsymbol{s}))\mathbf{V}(\mathbf{s'})$$

Si $\left(\left\| \mathbf{V_{tmp}} - \mathbf{V}(\mathbf{s}) \right\| \geq \boldsymbol{\varepsilon} \right)$ **Alors Quitte = faux**

$\mathbf{V}(\mathbf{s}) = \mathbf{V_{tmp}}$

$\mathbf{n} \leftarrow \mathbf{n} + \mathbf{1}$

9: **Pour** tout $\boldsymbol{s} \in \boldsymbol{S}$ **faire**

$$\boldsymbol{\pi}_{\mathbf{n+1}}(\mathbf{s}) \in \mathbf{argmax}_{\mathbf{a} \in \mathbf{A}} \left\{ \mathbf{r}(\mathbf{s}, \mathbf{a}) + \boldsymbol{\gamma} \sum_{\mathbf{s'} \in \mathbf{S}} \mathbf{p}(\mathbf{s'}|\mathbf{s}, \mathbf{a})\mathbf{V_n}(\mathbf{s'}) \right\}$$

10: Jusqu'à $\boldsymbol{\pi}_{\mathbf{n+1}} = \boldsymbol{\pi}_{\mathbf{n}}$

11: Retourner $\mathbf{V_n}, \boldsymbol{\pi}$

Figure 7 : Algorithme d'itération de la politique modifié, critère γ-pondéré

2.3. Programmation linéaire

En mathématique, un problème d'optimisation (encore appelé problème de programmation mathématique) consiste à trouver, parmi un ensemble donné un élément qui minimise (ou maximise) une fonction donnée tout en respectant d'éventuelles contraintes.

Déterminer un point maximisant ou minimisant une fonction donnée est un problème mathématique très commun, il peut se manifester sous différentes formes et présenter des degrés de difficulté très variés, selon les propriétés de la fonction (fonction d'une ou plusieurs variables, bornée, continue, convexe, dérivables…) ou de celle de son domaine de définition (ensemble discret, compact, connexe, convexe …). Même les cas les plus simples à première vue peuvent être la source de problèmes lorsqu'il s'agit de proposer des méthodes de résolution efficaces.

Plus formellement, l'optimisation est l'étude des problèmes qui s'expriment de la manière suivante :

Etant donné une fonction $\boldsymbol{f}: \boldsymbol{E} \rightarrow \mathbf{F}$ (**F** est un ensemble ordonné $(\boldsymbol{F}, \geq)$) trouver un élément de **E** tel que $\boldsymbol{f}(\tilde{\boldsymbol{x}}) \leq \boldsymbol{f}(\boldsymbol{x})$

2.2.1 Optimisation linéaire

Un problème de programmation linéaire (**PPL**) est un problème qui consiste à minimiser (ou maximiser) une fonction linéaire sur un domaine défini par une collection finie des contraintes linéaire.

Formalisation : La formalisation d'un programme est une tâche délicate mais essentielle car elle conditionne le découvert ultérieur de la bonne solution. Elle comporte les mêmes phases quelles que soient les techniques requises ultérieurement pour le traitement (résolution).

a. La détection du problème et l'identification des variables.
b. La formulation de la fonction objectif, et la exprimées sous la forme d'une fonction linéaire des variables identifiées.
c. La formulation des contraintes.

La forme standard d'un **PPL** s'écrit de la forme suivante :

$$(P_s): \begin{Bmatrix} \min(Z = C^T X) \\ S/C\ AX\ = b \\ X \geq 0 \end{Bmatrix}$$

<u>Notation :</u> $\boldsymbol{X} \geq \mathbf{0} \sim \boldsymbol{X} = (\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n})$ tel que $\boldsymbol{x_1}, \boldsymbol{x_2}, \dots, \boldsymbol{x_n} \geq \mathbf{0}$

Où

$$\boldsymbol{C} = \begin{pmatrix} \boldsymbol{c_1} \\ \boldsymbol{c_2} \\ \vdots \\ \boldsymbol{c_n} \end{pmatrix} : vecteur\ des\ coûts\ unitaires; \boldsymbol{b} = \begin{pmatrix} \boldsymbol{b_1} \\ \boldsymbol{b_2} \\ \vdots \\ \boldsymbol{b_n} \end{pmatrix} : vecteur\ second\ membres$$

$\boldsymbol{A} = (\boldsymbol{a_{ij}})_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n}}$: Matrice des contraintes

$\boldsymbol{Z}$: Fonction objectif à minimiser.

Remarque :

- Tout problème d'optimisation linéaire peut être écrit sous forme standard.
- Un **PPL**, peut s'écrit sous la forme suivante :

$$(\mathbf{P_c}): \begin{cases} \mathbf{min}(\mathbf{Z} = \mathbf{C^T X}) \\ \mathbf{S/C\ AX} \leq \mathbf{b} \\ \mathbf{X} \geq \mathbf{0} \end{cases}$$ Cette forme est dite forme canonique

2.2.2 Résolution d'un problème de la programmation linéaire

Etant donné un problème de programmation linéaire (P_s) standard :

$$(P_s): \begin{cases} \min(f(X) = C^T X) \\ S/C \quad AX = b \\ X \geq 0 \end{cases}$$

Notons que $\mathbf{E} = \{\mathbf{X} \in \mathbb{R}^{\mathbf{n}} \mid \mathbf{AX} = \mathbf{b}\ ,\ \mathbf{X} \geq \mathbf{0}\}$ l'ensemble admissible $\mathbf{A} \in \mathbf{M_{m,n}}(\mathbb{R})$

Remarque :

- Si $m > n$ le système $AX = b$ risque d'être surdéterminé (trop de conditions)
- Si $m = n$ alors la matrice A est carrée, cela revient généralement à résoudre un système linéaire d'équations.
  - $\mathbf{det}(\mathbf{A}) \neq \mathbf{0}$ Par Gramer $\begin{cases} \mathbf{E} = \{\mathbf{X}^* \geq \mathbf{0}\} \\ \mathbf{E} = \emptyset \textbf{ si } \mathbf{X}^* \textbf{ est non positive} \end{cases}$
  - $\mathbf{det}(\mathbf{A}) = \mathbf{0}$ Et $\mathbf{rg}(\mathbf{A}) < \mathbf{n}$ alors il existe des contraintes redondantes.
- Si $\mathbf{m} < \mathbf{n}$ avec $\mathbf{rg}(\mathbf{A}) = \mathbf{m}$ absence de redondance.

Dans le cadre de programme linéaire :

$\mathbf{A} \in \mathbf{M_{m,n}}(\mathbb{R})$ Avec $\mathbf{m} << \mathbf{n}$ (cas des problèmes réels de l'industrie et de l'économie).

Propriété : $\mathbf{rg}(\mathbf{A}) = \mathbf{m} \Leftrightarrow \exists$ une sous matrice carrée $\mathbf{M} \in \mathbf{M_m}(\mathbb{R})$ inversible de $\mathbf{A}$. $\mathbf{M}$ Est appelée une **base** du **PPL.**

Ensemble admissible :

L'ensemble $\mathbf{E} = \{\mathbf{X} \in \mathbb{R}^{\mathbf{n}} \mid \mathbf{AX} = \mathbf{b}\ ,\ \mathbf{X} \geq \mathbf{0}\}$ est appelé un **polytope** de $\mathbb{R}^{\mathbf{n}}$**.**

Si **E** est borné alors **E** est appelé **polyèdre** de $\mathbb{R}^{\mathbf{n}}$**.**

Idée de résolution de PPL

Puisqu'une solution de **PPL** (P_s): $\begin{cases} \min(f(X) = C^T X) \\ S/C \quad AX = b \\ \quad X \geq 0 \end{cases}$, se situe sur un sommet de

$\mathbf{E} = \{\mathbf{X} \in \mathbb{R}^{\mathbf{n}} \mid \mathbf{AX} = \mathbf{b}\,,\ \mathbf{X} \geq \mathbf{0}\}$

Une première méthode consiste à calculer tous les sommets de $\mathbf{E}\ \mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_p}$ et considérer :

$\mathbf{f}(\mathbf{X}^*) = \mathbf{min}\{\mathbf{f}(\mathbf{X_i}), \mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots, \mathbf{p}\}$ Ou $\boldsymbol{X}^*$ est une solution de $(\mathbf{P_s})$.

<u>Remarque :</u>

Le nombre de sommets à calculer est l'ordre majoré par $\mathbf{C_n^m}$, donc il faut résoudre $\mathbf{C_n^m}$ système ce qu'est très cher au niveau de calculs.

2.2.3 Méthodes de résolution d'un problème de programmation linéaire

2.2.3.1 Résolution graphique

Soit $\begin{array}{lll} f & : & \mathbb{R}^2 \to \mathbb{R} \\ (x_1, x_2) & \to & f(X) = C_1 x_1 + C_2 x_2 \end{array}$

<u>Idée :</u> à partir d'un **PPL :**

$$(\mathbf{P}): \begin{cases} \mathbf{min}(\mathbf{f}(\mathbf{X}) = \mathbf{C_1 x_1} + \mathbf{C_2 x_2}) \\ \mathbf{S/C} \begin{cases} \mathbf{C_{11} x_1} + \mathbf{C_{12} x_2} \geq \boldsymbol{b_1} \\ \mathbf{C_{21} x_1} + \mathbf{C_{22} x_2} \leq \boldsymbol{b_2} \\ \cdots \quad \cdots \\ \mathbf{C_{i1} x_1} + \mathbf{C_{i2} x_2} \geq \boldsymbol{b_i} \end{cases} \\ \mathbf{x_1}, \mathbf{x_2} \geq \mathbf{0} \end{cases}$$

Soit $\mathbf{a_{i1} x_1} + \mathbf{a_{i2} x_2} \geq \boldsymbol{b_i} \Rightarrow (\boldsymbol{\Delta_i}: \mathbf{a_{i1} x_1} + \mathbf{a_{i2} x_2} = \boldsymbol{b_i})$: une droite qui divise le plan en $^{\mathbf{1}}/_{\mathbf{2}}$ plans.

- On trace l'ensemble réalisable ou l'ensemble des solutions admissibles
  $\mathbf{E} = \{\mathbf{X} \in \mathbb{R}^{\mathbf{n}} \mid \mathbf{AX} = \mathbf{b}\,,\ \mathbf{X} \geq \mathbf{0}\}$ Telle que $\mathbf{E}$ est l'intersection de ces demi plans avec $\mathbf{E}$ est en dessous ou au-dessus de $\boldsymbol{\Delta_i}$.

2.2.3.2 Méthode Simplexe

La méthode du simplexe a été mise au point par Georges Dantzig dans les années **1950**. Elle se base sur une heuristique qui permet dans une très grande majorité des cas d'atteindre l'optimisation du problème très rapidement, son principe consiste à parcourir un chemin sur les frontières du domaine des solutions admissible, en s'approchant à chaque étape un peu plus de l'optimisation. L'heuristique essaye de trouve parmi tous les chemins qui soit court.

Idée et principe

- Partir d'une solution de base admissible (solution initiale) $\boldsymbol{X_0}$.
- Construire une solution de base admissible de $\boldsymbol{X_0}$ telle qu'une solution de base qui diminue la valeur de la fonction objectif $\mathbf{f}(\mathbf{X_1}) < \mathbf{f}(\mathbf{X_0})$ et ainsi de suite $\mathbf{X_1} \to \mathbf{X_2} \to \mathbf{X_3} \to \ldots \to \boldsymbol{X}^*$,

Le simplexe converge généralement en un nombre fini d'itération (en absence de cycle)

$\mathbf{f}(\mathbf{X}^*) = \mathbf{min}_{\mathbf{X} \in \mathbf{E}} \mathbf{f}(\mathbf{X})$.

Les étapes :

Soit le **PPL** standard (P) :

$$(P):\begin{cases}\min(f(X) = C^T X)\\ S/C \quad AX = b\\ \quad X \geq 0\end{cases}$$

Solution initiale : $\mathbf{X_0}\begin{pmatrix}\mathbf{X_M} \geq \mathbf{0}\\ \mathbf{X_N} = \mathbf{0}\end{pmatrix}$ **(**solution de base admissible).

On a donc : $\mathbf{f(X) = C^T X = f_0 + C_{m+1}x_{m+1} + \ldots + C_n x_n}$ où $\mathbf{C_{m+1}x_{m+1} + \ldots + C_n x_n}$ **:** sont des variables hors-base, et $f(X_0) = f_0$.

Le problème devient :

$$(\mathbf{P}):\begin{cases}\mathbf{min(f(X) = f_0 + C_{m+1}x_{m+1} + \ldots + C_n x_n)}\\ \mathbf{S/C}\begin{cases}\mathbf{1x_1 + 0x_2 + \ldots + 0x_m + a_{1,m+1}x_{m+1} + \ldots + a_{1,n}x_n = b_1}\\ \mathbf{0x_1 + 1x_2 + \ldots + 0x_m + a_{2,m+1}x_{m+1} + \ldots + a_{2,n}x_n = b_2}\\ \quad\cdots\quad\cdots\\ \mathbf{0x_1 + 0x_2 + \ldots + 1x_m + a_{m,m+1}x_{m+1} + \ldots + a_{m,n}x_n \geq b_m}\end{cases}\\ \mathbf{X \geq 0}\end{cases} \qquad (*)$$

- Si $\forall \mathbf{s} \in \{\mathbf{m+1}, \ldots, \mathbf{n}\}$, on a $\mathbf{C_s \geq 0}$: coût associé aux variables hors-base $\boldsymbol{x_s}$, alors toute augmentation de x_s va augmenter la valeur de **f**.
- Si $\exists\ \mathbf{s}\ /\ \mathbf{C_s < 0}$, alors toute augmentation de $\mathbf{x_s}$ va diminue **f**.

Remarque

Le problème précédent $\mathbf{AX = b} \Leftrightarrow (\mathbf{MN})\begin{pmatrix}\mathbf{X_M}\\ \mathbf{X_N}\end{pmatrix} = \mathbf{b}$

$$\Leftrightarrow \mathbf{MX_M + NX_N = b}$$

$$\Leftrightarrow \mathbf{I_m X_M + M^{-1}NX_N = M^{-1}b}$$

$$\Leftrightarrow \mathbf{I_m X_M + N' X_N = b}$$

$$\Leftrightarrow (\mathbf{I_m N'})\begin{pmatrix}\mathbf{X_M}\\ \mathbf{X_N}\end{pmatrix} = \mathbf{b}$$

Où : $\mathbf{M^{-1}N = N'}, \mathbf{M^{-1}b = b}$ **et** $\mathbf{I_m}$ **:** ***est la matrice identité***.

Soit $Z = f(X) = C^T X$ (en fonction des variables hors-base).

	x_1	x_2	…	x_r	…	x_m	x_{m+1}	…	x_s	…	x_n		
	0	0	…	0	…	0	C_{m+1}	…	C_s	…	C_n	Z	
x_1	1	0	$\cdots$	0	$\cdots$	0	$a_{1,m+1}$	$\cdots$	$a_{1,s}$	$\cdots$	$a_{1,n}$	b_1	L_1
x_2	0	1	$\cdots$	0	$\cdots$	0	$a_{2,m+1}$	$\vdots$	$a_{2,s}$	$\vdots$	$a_{2,n}$	b_2	L_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$		$\cdots$	$\vdots$	$\vdots$
x_r	0	0	$\cdots$	1	$\cdots$	0	$a_{r,m+1}$	$\vdots$	$a_{r,s}$	$\vdots$	$a_{r,n}$	b_r	L_r
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$			$\vdots$	$\vdots$
x_m	0	0	$\cdots$	0	$\cdots$	1	$a_{m,m+1}$	$\cdots$	$a_{m,s}$	$\cdots$	$a_{m,n}$	b_m	L_m

L_i : Les lignes du tableau.

Solution actuelle : $\begin{cases} \boldsymbol{x_1 = b_1} \\ \boldsymbol{x_2 = b_2} \\ \vdots \\ \boldsymbol{x_m = b_m} \end{cases}$ $\qquad \boldsymbol{x_{m+1} = x_{m+2} = \ldots = x_n = 0}.$

Variables en base $\qquad$ Variable hors-base

Pour entrer $\boldsymbol{x_s}$ à la place de $\boldsymbol{x_r}$, on effectue un <u>pivotage de Gauss</u> de tableau par rapport au pivot $\boldsymbol{a_{rs}}$:

1) Diviser la ligne $(\mathbf{L_r})$ par le pivot $\boldsymbol{a_{rs}}$ **i.e** $\mathbf{(1/a_{rs})L_r \rightarrow L_p}$ ou $\boldsymbol{L_p}$: ligne pivot.
2) Pour $\mathbf{i = 1, 2, \ldots, m(i \neq r)}$, on a : $\mathbf{L_i \rightarrow L_i - a_{is}L_p}$ **.**
3) Le tableau devient :

	x_1	x_2	…	x_r	…	x_m	x_{m+1}	…	x_s	…	x_n		
	0	0	…	C'_r	…	0	C'_{m+1}	…	0	…	C'_n	Z	
x_1	1	0	$\cdots$	$a'_{1,r}$	$\cdots$	0	$a'_{1,m+1}$	$\cdots$	0	$\cdots$	$a'_{1,n}$	b'_1	
x_2	0	1	$\cdots$	$a'_{2,r}$	$\cdots$	0	$a'_{2,m+1}$	$\vdots$	0	$\vdots$	$a'_{2,n}$	b'_2	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$		$\cdots$	$\vdots$	
x_r	0	0	$\cdots$	$a'_{r,r}$	$\cdots$	0	$a'_{r,m+1}$	$\vdots$	1	$\vdots$	$a'_{r,n}$	b'_r	L_p
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$		$\vdots$		$\vdots$	$\vdots$	
x_m	0	0	$\cdots$	$a'_{m,r}$	$\cdots$	1	$a'_{m,m+1}$	$\cdots$	0	$\cdots$	$a'_{m,n}$	b'_m	

Solution actuelle : $\begin{cases} \boldsymbol{x_1 = b'_1} \\ \boldsymbol{x_2 = b'_2} \\ \vdots \\ \boldsymbol{x_s = b'_s} \\ \vdots \\ \boldsymbol{x_m = b'_m} \end{cases}$ $\qquad \boldsymbol{x_{m+1} = x_{m+2} = \ldots = x_n = 0}.$

Variables en base $\qquad$ Variable hors-base

Et $\mathbf{Z - Z_0 = 0 \Rightarrow Z = Z_0}$

Nous allons montrer maintenant un organigramme de la méthode simplexe :

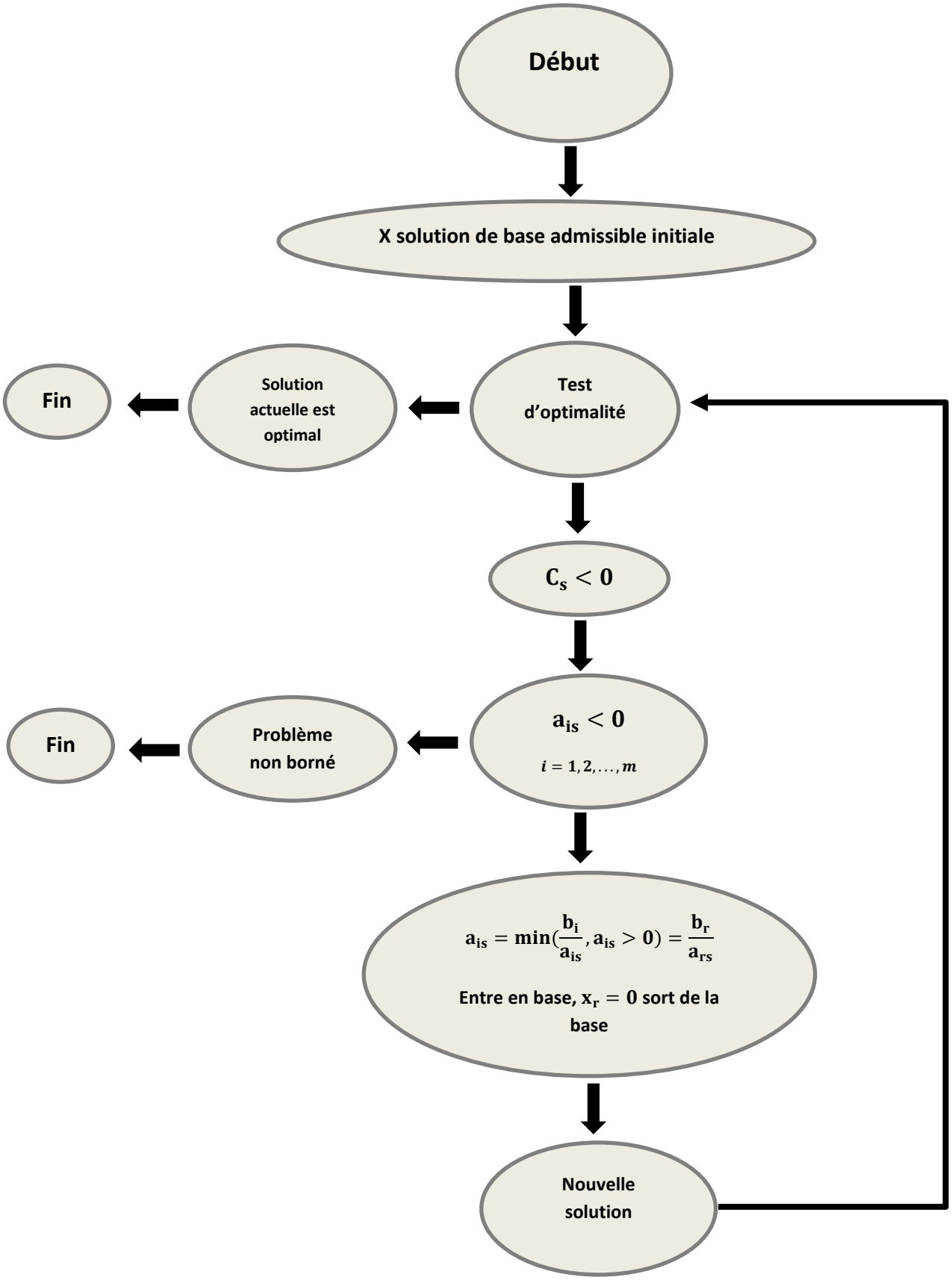

Figure 8 : Organigramme de la méthode simplexe.

Il y a beaucoup d'autres méthodes d'optimisation, comme le simplexe comme la **Méthode des deux-phases (Gauss)** et aussi la **Méthode du grand M (Big M)**.

2.2.4 Quelques problèmes de la programmation linéaire

Les problèmes de transport interviennent dans les nombreux domaines de l'optimisation. D'autre manière générale, on entendra par problème de transport tout problème d'optimisation du transfert entre des points origines qu'on appelle des « sources » vers des points de destination appelés des « puits ». Un problème de transport est un programme de minimisation de coût transport, il est ainsi, un problème qui s'inscrit dans la famille des programmes linéaire, qui a comme but, l'acheminement au moindre des marchandises depuis **m** sources vers **n** destinations.

Problème du voyageur du commerce : Il existe un type de problème, qu'est le problème d'un voyageur commerce, qui doit visiter **n** villes (noté sommet) en passant par chaque ville exactement une fois, il commence par une ville quelconque et il termine en retournant à la ville départ, les distance entre les villes sont connues, il faut trouver le chemin qui minimise la distance parcoure. La notion de distance peut être remplacée par d'autres notions comme le temps qu'il met ou l'argent qu'il dépensera, en général on parle de coût.

Problème de coloration : Les problèmes de coloration des graphes, pour un graphe non orienté **G**, consiste à attribuer une couleur à chaque sommet de telle sorte qu'une même couleur ne soit pas attribué à deux sommets adjacents (relié par une arrête). Ce problème est intéressant à plusieurs titres. Premièrement, en raison de la simplicité de son énoncé et de sa complexité théorique. Deuxièmement par ce qu'il apparait naturellement dans de multiple problème pratique : La confection d'horaire d'examens dans une université, les emplois de temps, répartition des entités (dans les bases des données), colorier des cartes géographiques et finalement les problèmes d'optimisation avec contrainte dans les réseaux, ainsi que plusieurs d'autre exemples.

2.4. Algorithme d'itération de la valeur et ses variantes

L'algorithme d'itération de la valeur est le plus largement utilisé et le mieux compris pour la résolution de problèmes de décision de Markoviens. Vous le connaissez peut-être déjà sous d'autres noms, notamment des approximations successives, une sur relaxation, une induction en arrière, une itération pré-Jacobi ou même une programmation dynamique. L'attrait de cet algorithme peut peut-être être attribué à sa simplicité conceptuelle, à sa facilité de codage et de mise en œuvre, et à sa similitude avec les approches utilisées dans d'autres domaines des mathématiques appliquées, en plus de fournir un outil numérique simple pour résoudre ces modèles, il peut également être utilisé pour obtenir des résultats concernant la structure des politiques optimales.

Cette section commence par présenter l'algorithme d'itération de valeur dans sa forme la plus basse. Nous introduisons ensuite plusieurs variantes qui améliorent la convergence.

Dans cette section, nous supposons que le maximum indiqué en **2.5** ou le même équivalent en **2.6** est atteint pour tous les $\boldsymbol{v} \in \boldsymbol{V}$, ce qui se produit, par exemple, lorsque $\boldsymbol{A_s}$ est fini pour chaque $\boldsymbol{s \epsilon S}$, nous n'avons pas besoin de cette hypothèse pour établir la convergence. L'algorithme, cependant, nous l'adoptons pour simplifier l'exposé. Il inclut presque tous les modèles que nous pourrions résoudre de manière numérique. Par conséquent, nous nous intéresserons à la résolution du système d'équations.

$$U(s) = max_{a \in A}\{ r(s,a) + \alpha \sum_{S' \in \delta} P(s',a,s)\, U(s')\} \qquad 2.1$$

Nous commençons cette section par une discussion de quelques concepts généraux concernant les vitesses de convergence des algorithmes.

Soit $\{y_n\} \subset V$ une séquence qui converge vers y^*, qu'est : $\lim_{n\to\infty} ||y_n - y^*|| = 0$. On dit que $\{y_n\}$ converge vers à l'ordre $\alpha, \alpha > 0$ si et seulement s'il existe une constante $K > 0$ qui vérifie :

$$||y_n - y^*|| \leq K||y_n - y^*||^{\alpha} \qquad 2.2$$

Pour $n = 1, 2, \ldots$, la convergence linéaire correspondante (avec $\alpha < 1$, on dit que $\{y_n\}$ converge super-linéairement lorsque :

$$\limsup_{n\to\infty} \frac{||y_{n+1} - y^*||}{||y_n - y^*||} = 0 \qquad 2.3$$

Notez qu'une séquence super-linéairement convergente se converge plus rapidement qu'une séquence linéairement convergente.

Si $\{y_n\}$ convergent en ordre α, nous définissons le taux de convergence $\widehat{K}$ comme étant le plus petit K tel que 6.3.1 soit valable pour tout n. Bien sûr, plus le taux de convergence est faible, plus y_n qui converge rapidement vers y^*. Le plus souvent, nous utilisons le taux de convergence pour comparer des séquences linéairement convergentes. L'exigence que 6.3.1 soit valable pour tous les n qui peut indiquer que cette mesure est insensible aux propriétés de convergence réelles de la séquence. Pour obtenir une meilleure vue d'ensemble de la vitesse de convergence, définissez le *taux de convergence moyen asymptotique* (TCMA) comme suit :

$$\limsup_{n\to\infty} \left[\frac{||y_n - y^*||}{||y_0 - y^*||}\right]^{\frac{1}{n}} \qquad 2.4$$

Bien sûr on suppose que $y_0 \neq y^*$. Notez que cette définition peut être étendue aux séries qui convergent à un ordre supérieur à 1 en augmentant la quantité du dénominateur à la puissance α^n.

2.4.1. Algorithme d'itération de la valeur

Comme nous l'avons signalé au chapitre précèdent, l'étape (**4**) clé de l'algorithme, peut être exprimé en notation vectorielle comme le suivant :

$$V^{n+1} = L.V^n \qquad 2.8$$

Le théorème suivant fournit les principaux résultats concernant la convergence de l'algorithme d'itération de la valeur :

Théorème 2.1

Soit $\boldsymbol{V_0} \in \boldsymbol{V}$, $\boldsymbol{\varepsilon} > \mathbf{0}$ et soit $\{\boldsymbol{V^n}\}$ satisfait l'étape **(7)**, alors pour tout $\boldsymbol{n} \geq \boldsymbol{N}$ on a :

- **A.** V^n Converge en norme vers V^*
- **B.** N est fini pour lequel **(6)** est valable pour tout $\boldsymbol{n} \geq \boldsymbol{N}$
- **C.** La politique stationnaire $\boldsymbol{\pi}$ définie en **(7) est $\boldsymbol{\varepsilon}$-optimale** avec
- **D.** $\left|\left|\boldsymbol{V_{n+1}} - \boldsymbol{V^*}\right|\right| < \boldsymbol{\varepsilon}$ Chaque fois ou **(6)** est vérifié

Preuve

Les deux points **A** et **B**, sont obtenus immédiatement d'après l'équation $(\mathbf{1.4.2})$, supposons maintenant que **(2.3.6)** est vérifié pour certain $\boldsymbol{n}$, et $\boldsymbol{\pi_\varepsilon}$satisfait **(2.3.7)**. Ainsi,

$$\left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^*}\right|\right| \leq \left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| + \left|\left|\boldsymbol{V_\alpha^{n+1}} - \boldsymbol{V_\alpha^{\pi^\infty}}\right|\right| \qquad \mathbf{2.9}$$

Notons que $\boldsymbol{V_\alpha^{\pi^\infty}}$ est un point fixe de $\boldsymbol{L_{\pi_\varepsilon}}$**,** et comme conséquence de **(2.7)**, $\boldsymbol{L_{\pi_\varepsilon} V^{n+1}} = \boldsymbol{L V^{n+1}}$**.**

D'après l'équation **(2.9)**, en prenant le terme qu'est à droit, on aurait :

$$\begin{aligned}
\left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| &= \left|\left|\boldsymbol{L_{\pi_\varepsilon} V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| \\
\left|\left|\boldsymbol{L_{\pi_\varepsilon} V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| &\leq \left|\left|\boldsymbol{L_{\pi_\varepsilon} V_\alpha^{\pi^\infty}} - \boldsymbol{L V_\alpha^{n+1}}\right|\right| + \left|\left|\mathbf{L}\boldsymbol{V_\alpha^{n+1}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| \\
&= \left|\left|\boldsymbol{L_{\pi_\varepsilon} V_\alpha^{\pi^\infty}} - \boldsymbol{L_{\pi_\varepsilon} V_\alpha^{n+1}}\right|\right| + \left|\left|\mathbf{L}\boldsymbol{V_\alpha^{n+1}} - \boldsymbol{L V_\alpha^{n}}\right|\right| \\
&= \boldsymbol{\alpha}\left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^{n+1}}\right|\right| + \boldsymbol{\alpha}\left|\left|\boldsymbol{V_\alpha^{n+1}} - \boldsymbol{V_\alpha^{n}}\right|\right|
\end{aligned}$$

Notons que la dernière équation est obtenue de la faite que : $\boldsymbol{L}\ et\ \boldsymbol{L_{\pi_\varepsilon}}$ sont contractantes, ainsi on aura le terme suivant :

$$\left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V^{n+1}}\right|\right| \leq \frac{\boldsymbol{\alpha}}{\mathbf{1} - \boldsymbol{\alpha}}\left|\left|\boldsymbol{V^{n+1}} - \boldsymbol{V^n}\right|\right|$$

La partie à droit dans l'équation $(\mathbf{2.9})$ implique l'inégalité suivante :

$$\left|\left|\boldsymbol{V^{n+1}} - \boldsymbol{V_\alpha^*}\right|\right| \leq \frac{\boldsymbol{\alpha}}{\mathbf{1} - \boldsymbol{\alpha}}\left|\left|\boldsymbol{V^{n+1}} - \boldsymbol{V^n}\right|\right|$$

Ainsi, et lorsque **(6)** est vérifier, et notons que les deux éléments de la partie droite de l'équation $(\mathbf{2.9})$ sont majorés par : $\boldsymbol{\varepsilon}/\mathbf{2}$, alors on a :

$$\left|\left|\boldsymbol{V_\alpha^{\pi^\infty}} - \boldsymbol{V_\alpha^*}\right|\right| \leq \boldsymbol{\varepsilon} \qquad \mathbf{2.10}$$

Ainsi, cette équation établit les deux parties **C** et **D**. D'où théorème.

Ainsi, l'algorithme d'itération de valeur trouve une stratégie stationnaire optimale dans un nombre fini d'itérations. Bien sûr, $\boldsymbol{\pi}_{\boldsymbol{\alpha}}^{\infty}$ pourrait être optimal, mais l'algorithme décrit ci-dessus ne fournit aucun moyen de le déterminer. En combinant cet algorithme avec des méthodes d'élimination des actions, nous pouvons souvent nous assurer que l'algorithme se termine avec une stratégie optimale. En pratique, choisissez un $\boldsymbol{\varepsilon}$ suffisamment petit, nous assurer que l'algorithme s'arrête avec une stratégie très proche de l'optimal. Mais le choix d'un $\boldsymbol{\alpha}$ très grand peut empêcher l'algorithme de convergé rapidement, nous allons monter un exemple avec un $\boldsymbol{\alpha = 0,95}$, ainsi l'algorithme a convergé dans **268** itérations, comme le suivant :

Tableau 1 : Les valeurs du vecteurs gains optimale V, de l'algorithme d'itération de la valeur dans chaque itération

N	$V^n(S_1)$	$V^n(S_1)$	$V^n(S_1)$	$\lVert V^{n-1} - V^n \rVert$
0	15,000000	22,000000	19,000000	22,000000
1	33,145000	38,625000	36,764999	18,145000
2	49,769051	55,393452	53,425152	16,768452
3	65,620201	71,230881	69,272873	15,851151
4	80,673248	86,285240	84,326248	15,054359
5	94,974182	100,586037	98,627136	14,300934
6	108,559998	114,171883	112,212975	13,585846
7	121,466553	127,078415	125,119514	12,906555
8	133,727753	139,339630	137,380722	12,261215
9	145,375916	150,987778	149,028870	11,648163
10	156,441650	162,053528	160,094620	11,065750
20	240,806870	246,418747	244,459839	6,625473
30	291,319427	296,931274	294,972412	3,966919
40	321,563171	327,175018	325,216125	2,375153
50	339,671204	345,283081	343,324188	1,422089
60	350,513184	356,125000	354,166138	0,851471
70	357,004669	362,616547	360,657623	0,509827
80	360,891357	366,503235	364,544312	0,305237
90	363,218445	368,830292	366,871399	0,182770
100	364,611786	370,223663	368,264771	0,109436
120	365,945496	371,557343	369,598450	0,039246
140	366,423584	372,035461	370,076538	0,014069
160	366,595032	372,206909	370,247986	0,005066
180	366,656464	372,268341	370,309418	0,001801
200	366,678497	372,290375	370,331451	0,000671
220	366,686310	372,298218	370,339294	0,000244
240	366,689178	372,301025	370,342133	0,000092
260	366,690186	372,302032	370,343140	0,000031
261	366,690216	372,302063	370,343170	0,000031
262	366,690247	372,302124	370,343201	0,000061
263	366,690277	372,302124	370,343231	0,000031
264	366,690308	372,302185	370,343262	0,000061
265	366,690308	372,302216	370,343292	0,000031
266	366,690338	372,302216	370,343323	0,000031
267	366,690369	372,302246	370,343323	0,000031
268	366,690369	372,302246	370,343323	0,000000

Les conclusions du théorème précèdent, ne sont pas limitées aux modèles avec des espaces d'états discrets et des ensembles d'actions finis. La convergence de l'itération de valeur est assurée lorsque **L** est une application contractante sur **V**. Ceci signifie que l'itération de la valeur convergera normalement (en norme), même lorsque **S** est compact ou Borel. Malheureusement, l'évaluation numérique de la maximisation en **(2.3.5)** n'est pratique que lorsque **S** est fini. Pour des espaces d'états plus généraux, la maximisation ne peut être effectuée qu'en utilisant une structure spéciale des fonctions de récompense, de probabilité de transition et une fonction de valeur pour laquelle on cherche à déterminer la structure de maximisation des règles de décision, comme le cas des **MDP** à horizon fini. Les alternatives générales incluent la discrétisation et / ou la troncature.

Complexité de l'algorithme d'itération de la valeur

L'efficacité de l'algorithme dépend de deux facteurs : la complexité d'une itération et le nombre d'itération nécessaire pour la convergence. Chaque itération consiste à calculer la valeur de transition entre les états, pour chaque action : cela nécessite $|\boldsymbol{A}||\boldsymbol{S}|^2$ opération. Le nombre d'itérations nécessaire est plus difficile à déterminer. D'après la forme de la matrice canonique de la matrice de transition, on conclut que l'algorithme est polynomial selon $|\boldsymbol{A}|, |\boldsymbol{S}|, \boldsymbol{\alpha}\ et\ \boldsymbol{B},$ où $\boldsymbol{B}$ est le nombre de bits nécessaires à la représentation des données du problème.

2.4.2. Algorithme Gauss-Seidel Itération de la valeur

Avant de présenter l'algorithme d'itération de la valeur amélioré ou Gauss-Seidel Itération de la valeur, on signale que son point fort est dans l'utilisation immédiate, de la fonction de valeur d'un état juste après mise à jour, comme le suivant :

Répéter

$\boldsymbol{Pour\ tout\ i\ \in S\ Faire}$

$$\boldsymbol{V_n(i) = max_{a\in A(i)}\{ r_{ia} + \alpha \sum_{j<i} P_{iaj} V_n(j) + \sum_{j\geq i} P_{iaj} V_{n-1}(i)\}}$$

Jusqu'à ($\boldsymbol{Max_{i\in S}\left||V_{n-1} - V_n|\right| < \varepsilon}$ **)**

Notons que cette modification dans l'algorithme résulte une efficacité au niveau de complexité et au temps d'exécution, chose qu'on va le montrer dans le chapitre qui vient par la suite.

Nous allons maintenant faire une décomposition régulière pour le passage du l'étape **5** de l'algorithme, pour se faire, soit $\boldsymbol{\pi}$ une règle de décision obtenue après la maximisation selon les actions, lorsque **(a)** est vérifiée, pour $\boldsymbol{S_0, S_1 \ldots, S_{N-1}, S_N}$ et on écrit : $\boldsymbol{P_\pi = P_\pi^L + P_\pi^u}$,

Où :

$$\boldsymbol{P_\pi^{L-} = \begin{bmatrix} 0 & 0 & \ldots & \ldots & 0 \\ P_{21} & 0 & \ldots & \ldots & 0 \\ P_{31} & P_{32} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ P_{N1} & \ldots & \ldots & P_{NN-1} & 0 \end{bmatrix}} \text{ et } \boldsymbol{P_\pi^U = \begin{bmatrix} P_{11} & P_{12} & \cdots & \ldots & P_{1N} \\ 0 & P_{21} & \ldots & \ldots & P_{2N} \\ 0 & 0 & P_{33} & \cdots & P_{3N} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ldots & \cdots & P_{NN} \end{bmatrix}}$$

Cela signifie que l'étape **2** de l'algorithme peut s'écrit sous la forme suivante :

$$\mathbf{V^{n+1} = (I - \alpha P_\pi^L)^{-1}(\alpha P_\pi^U)V^n + (I - \alpha P_\pi^L)^{-1} r_\pi}$$

On pose maintenant :

$\mathbf{Q_\pi = (I - \alpha P_\pi^L)}$ et $\mathbf{R_\pi = \alpha P_\pi^U}$, alors l'algorithme peut être exprimé sous la forme suivante :

$$\mathbf{V^{n+1} = (Q_\pi)^{-1} R_\pi V^n + (Q_\pi)^{-1} r_\pi}$$

Notons qu'on a exprimé L'algorithme **GSVI** en termes d'inverse de $\mathbf{Q_\pi}$, ce n'est pas si important de l'expliquer en détails, mais c'est juste pour la résolution de l'étapes **2** de l'algorithme. Nous allons maintenant prouver un théorème de la convergence générale, de cette méthode qu'est basé sur la décomposition régulière.

Théorème 2.3.2

Supposons que $(\mathbf{Q_\pi}, \mathbf{r_\pi})$ est la décomposition régulière de $\mathbf{I - \alpha} \boldsymbol{P_\pi}$ **,** pour toute règle de décision $\boldsymbol{\pi}$, est que : $\boldsymbol{\alpha} \equiv \boldsymbol{sup_\pi} \left| \left| (\mathbf{Q_\pi})^{-1} \mathbf{R_\pi} \right| \right| < \mathbf{1}$

Alors,

- **a.** $\forall\, \boldsymbol{V^0 \epsilon\, V^{n+1} = max_\pi}\{(\mathbf{Q_\pi})^{-1}\mathbf{R_\pi V^n} + (\mathbf{Q_\pi})^{-1}\mathbf{r_\pi}\} \equiv \boldsymbol{T}\mathbf{V^n}$ Converge vers $\boldsymbol{V_\alpha^*}$**.**
- **b.** La quantité $\boldsymbol{V_\alpha^*}$ est point fixe de $\boldsymbol{T}$**.**
- **c.** La séquence $\{\boldsymbol{V^n}\}$ Convergez globalement avec le premier ordre à un taux inférieur ou égal à $\boldsymbol{\alpha}$, son taux de convergence moyen asymétrique global est inférieur ou égal à $\boldsymbol{\alpha}$ et converge globalement à $\boldsymbol{O(\beta^n)}$ où le $\boldsymbol{\beta \leq \alpha}$**.**

Nous allons lancer l'algorithme d'itération de la valeur modifié, sur le même espace état traité avant, pour voir la déférence, au niveau du nombre d'itérations.

Tableau 2 : Les valeurs du vecteurs gains optimale V, de l'algorithme d'itération de la politique dans chaque itération

N	$V^n(S_1)$	$V^n(S_1)$	$V^n(S_1)$	$\left\|\left\|V^{n-1} - V^n\right\|\right\|$
0	15,000000	30,549999	31,981750	31,981750
1	41,748081	57,541420	59,450573	27,468822
2	67,570465	82,562271	84,379044	25,822384
3	91,448685	105,671997	107,243423	23,878220
4	113,432388	126,961105	128,264648	21,983704
5	133,661774	146,556259	147,602707	20,229385
6	152,275543	164,587845	165,395081	18,613770
7	169,402557	181,179535	181,766022	17,127014
8	185,161499	196,445999	196,829224	15,758942
9	199,661682	210,493042	210,689209	14,500183
10	213,003601	223,418030	223,442047	13,341919
20	299,842163	307,542969	306,446594	5,803284
30	337,613892	344,134399	342,550659	2,524231
40	354,043396	360,050476	358,254761	1,097961
50	361,189606	366,973358	365,085449	0,477570

60	364,297974	369,984619	368,056580	0,207703
70	365,650024	371,294434	369,348938	0,090363
80	366,238098	371,864105	369,911072	0,039307
90	366,493896	372,111938	370,155579	0,017090
100	366,605164	372,219696	370,261902	0,007446
110	366,653534	372,266571	370,308136	0,003204
120	366,674561	372,286957	370,328247	0,001404
130	366,683716	372,295776	370,336975	0,000610
140	366,687622	372,299591	370,340729	0,000275
150	366,689453	372,301361	370,342468	0,000153
160	366,690247	372,302124	370,343231	0,000061
161	366,690308	372,302185	370,343262	0,000061
162	366,690308	372,302216	370,343323	0,000061
163	366,690369	372,302246	370,343323	0,000061
164	366,690369	372,302246	370,343323	0,000000

Notons que l'algorithme ici a convergé juste en **164** itérations, pendant que l'algorithme d'itération de la valeur a convergé en **268** itérations, en remarquant que les deux algorithmes ont convergé avec les mêmes valeurs de **V**.

2.4.3. Conclusion partielle

Pour ces deux derniers algorithmes, la résolution du système linéaire est la partie la plus couteuse alors que dans l'algorithme d'itération de la valeur (**VI**), il y a une résolution directe de ce système. Ce qui implique un grand nombre d'itérations, ainsi que le temps pris pour la convergence (comme nous avons obtenu dans l'exemple précèdent), par contre dans la version modifiée de l'algorithme d'itération de la valeur (**GSVI**), il y a une résolution itérative, d'où la convergence rapide de l'algorithme, avec moins d'itérations que l'algorithme classique.

2.5. Algorithme d'itération de la politique et ses variantes

L'itération de la politique ou l'approximation dans l'espace des politiques est une autre méthode de résolution des problèmes de décision de Markov à horizon fini. Bien que l'itération de valeur puisse être considérée comme une application d'une approche générale pour trouver des points fixes, l'itération de la politique semble directement liée à la structure particulière des problèmes de décision de Markov. Cela s'applique aux problèmes stationnaires à horizon fini. Ce n'est pas une approche efficace pour résoudre les problèmes de la fin des horizons.

La convergence de l'algorithme d'itération de la politique pour les modèles avec des ensembles d'états finis et d'actions. Les résultats en secondes 6 et 6 s'appliquent à tout modèle dans lequel, pour toute stratégie $\boldsymbol{\pi} \in \mathbf{D}$ tel que :

$$\boldsymbol{\pi} \in \boldsymbol{argmax}_{a \in A(i)}(\boldsymbol{r}_{\pi} + \boldsymbol{\alpha}\boldsymbol{P}_{\pi}\boldsymbol{V})$$

Cela équivalant à dire que, pour tout $\boldsymbol{v} \in \boldsymbol{V}$ et $\boldsymbol{s} \in \boldsymbol{S}$, il existe une action $\boldsymbol{a}_t^s \in \boldsymbol{A}_s$, qui vérifie :

$$\boldsymbol{\pi} \in \boldsymbol{argmax}_{a \in A(i)}\left\{\boldsymbol{r}_{ia} + \boldsymbol{\alpha}\sum_{j \in S}\boldsymbol{P}_{iaj}\boldsymbol{V}(j)\right\}$$

Cette généralisation inclut des modèles à états finis avec un espace d'actions compacts et des fonctions de récompense et de transition continues, des modèles à états comptables avec des jeux d'action compacts, des fonctions de récompense et de transition continues bornées, ainsi que des modèles plus généraux reposant sur des hypothèses appropriées.

2.5.1. Algorithme d'itération de la politique

Lorsque l'ensemble des règles stationnaires déterministes n'est pas fini, par exemple lorsque **S** est fini et **A** est compact ou **S** comptable, la preuve de l'étape (**5**) de l'algorithme d'itération de la politique, n'est plus valide car rien ne garantit que les critères d'arrêt de l'étape **7**, soient toujours satisfaits. Dans de tels cas, nous allons utiliser l'approche suivante pour démontrer la convergence.

a. Quelles sont les conséquences, si on débute la compétition entre les stratégies à l'étape 3 (sachant que l'étape **1** est vérifiée) en commençant avec un $\boldsymbol{v_0} \in \boldsymbol{V}$ arbitraire.

b. Quel est l'ordre et le taux de convergence de l'algorithme.

Il est préférable de répondre à ces questions en fournissant une représentation récursive des itérations de l'algorithme d'itération de la politique similaire à celle utilisée pour analyser une itération de la valeur. Nous commençons à le faire maintenant, par définir l'opérateur suivant : $\mathbf{B} : \mathbf{V} \rightarrow \mathbf{V}$ par,

$$\mathbf{B} \equiv \boldsymbol{argmax}_{\pi \in D}\{\boldsymbol{r_\pi} + (\boldsymbol{\alpha P_\pi} - \boldsymbol{I})\boldsymbol{v}\} \qquad \mathbf{2.13}$$

Par conséquence : $\mathbf{B} = \mathbf{Lv} - \mathbf{v}$. Alors l'équation d'optimalité peut être exprimée de la façon suivante :

$$\mathbf{Bv} = \mathbf{0} \qquad \mathbf{2.14}$$

En utilisant cette notation, alors la résolution de l'équation d'optimalité, revient à trouver un zéro de **B** destiné à un point fixe de $\boldsymbol{L}$.

Pour $\boldsymbol{v} \in \boldsymbol{V}$, posons $\boldsymbol{D_v}$ L'ensemble des règles de décision, **v**-améliorante. Alors $\boldsymbol{\pi_v} \in \boldsymbol{D_v}$, si :

$$\boldsymbol{\pi_v} \in \boldsymbol{argmax}_{\pi \in D}(\boldsymbol{r_\pi} + (\boldsymbol{\alpha P_\pi} - \boldsymbol{I})\boldsymbol{v}) \qquad \mathbf{2.15}$$

Notons que, l'identité **I** n'a pas d'effet sur la sélection d'une règle de décision de la maximisation.

La proposition suivante fournit la base de notre analyse « géométrique » de cet algorithme.

Proposition 2.3.3 Pour tout $\mathbf{u}, \mathbf{v} \in \mathbf{V}$ et pour tout, $\boldsymbol{\pi_v} \in \mathbf{D_v}$: $\mathbf{Bu} \geq \mathbf{Bv} + (\boldsymbol{\alpha}\mathbf{P}_{\pi_v} - \mathbf{I})(\mathbf{u} - \mathbf{v})$

Preuve

A partir de la définition de **B**,

$$\mathbf{Bu} \geq \mathbf{r}_{\pi_v} + (\alpha \mathbf{P}_{\pi_v} - \mathbf{I})\mathbf{u} \qquad 2.16$$

Et

$$\mathbf{Bv} = \mathbf{r}_{\pi_v} + (\alpha \mathbf{P}_{\pi_v} - \mathbf{I})\mathbf{v} \qquad 2.17$$

La soustraction du $(\mathbf{2.17})$ par $(\mathbf{2.16})$ donne le résultat souhaité.

Nous nous référons à l'inégalité $(\mathbf{2.16})$ de support et la considérons comme une généralisation de l'espace vectoriel de l'inégalité de gradient qui définit les fonctions convexes dans $\mathbb{R}^{\mathbf{n}}$. Ainsi, dans un sens généralisé, l'opérateur $\mathbf{B}$ est "convexe" et $\alpha \mathbf{P}_{\pi_v} - \mathbf{I}$ est le "support" de $\mathbf{B}$ en $\mathbf{v}$. Nous trouvons qu'il est bien de faire référence à $\mathbf{D}_{\mathbf{v}}$ en tant qu'ensemble de règles de décision sur $\mathbf{v}$.

2.6. Algorithmes parallèles

Bien que la puissance et la capacité mémoire des ordinateurs ne cessent de croître, certains utilisateurs désirent toujours obtenir leurs résultats plus rapidement. D'autres souhaitent aussi faire tourner des simulations de plus grande précision en conservant un temps de calcul raisonnable. Le coût de fabrication d'un processeur toujours plus puissant croît exponentiellement avec la vitesse avec laquelle il peut faire les calculs. Par contre obtenir **N** processeurs ne représente qu'un coût **N** fois plus grande que celui d'un seul processeur. Le coût total de la machine parallèle (incluant le réseau d'interconnexion...) reste donc du même ordre.

Le parallélisme consiste à utiliser plusieurs ressources disponibles, processeurs, mémoires, disques, etc., pour qu'elles participent ensemble au calcul d'une application.

En multipliant les ressources par N, un utilisateur peut espérer :

- calculer N fois plus vite,
- calculer des problèmes occupant N fois plus d'espace mémoire.

Dans le chapitre qui vient, nous traitons le problème de la minimisation du temps d'exécution, en profitant de l'indépendance du calcul de l'équation d'optimalité $(\mathbf{1.4.1})$, afin de pourrais proposer une approche d'une nouvelle version (parallèle) d'algorithme d'itération de la valeur, pour réduire le temps d'exécution.

2.7. Conclusion

L'étude théorique faite dans ce chapitre, nous permet de constater que tous ces solveurs semblent beaucoup plus adaptés au cas où l'espace d'états est fini. Elle nous a permis aussi d'identifier la différence entre les algorithmes étudiés, au niveau de la convergence, ou encore au niveau des preuves de l'existence de la solution. Dans le chapitre suivant, on fera une étude comparative de ces solveurs, au niveau de temps d'exécution, en changeant chaque fois, un paramètre.

Chapitre III : Etude comparatives des solveurs

3.1. Introduction

Dans ce chapitre nous allons faire une étude comparative entres les solveurs, ou on va comparer chaque algorithme avec ces variantes. Comme nous l'avons vu dans le chapitre précédent, ont fait une comparaison théorique, au niveau de complexité de ces solveurs, alors maintenant la comparaison serait faite au niveau de temps d'exécution. Une interprétation graphique de ces résultats, serai présenter juste après, pour pourrait, enfin sortir avec des décisions sur la performance de chaque algorithme.

L'implémentation de tous ces algorithmes est faite avec le langage **C++,** et le choix de ce langage, est dû à sa forte précision sur les calculs du temps d'exécution, avec un espace d'état assez grand, qui va jusqu'à $\mathbf{10^5}$ pour qu'on puisse remarquer la différence entre les algorithmes.

Notons que, la matrice de transition, et puisqu'elle est stochastique, alors elle doit vérifier quelque condition, ce qu'implique qu'on doit le prendre en considération aussi. Alors pendant la construction de l'espace d'état, on doit assurer que la somme des lignes de la matrice de transition vaut bien à **1**, c'est-à-dire, la somme des probabilités de transition de chaque état, sur son ensemble de successeurs vaut **1**. Ainsi, que la généralisation de ces successeur (pour chaque état), les récompenses, l'ensemble d'action (dans chaque état) sont tous aléatoires

La comparaison sera faite au niveau de temps d'exécution et le nombre d'itération, et bien sûr, sur des espaces d'état très grands, afin de remarquer l'efficacité de chacun de ces algorithmes.

3.2. Algorithme d'itération de la valeur et ses variantes

Dans cette section nous intéressant aux deux algorithmes suivants : l'algorithme d'itération de la valeur, et ces variantes, nous allons comparer les algorithmes suivants : l'itération de la valeur **(VI)**, l'itération de la valeur Jacobi (**VI_J**), itération de la valeur- Gauss-Seidel **(GSVI)** et itération de la valeur-Gausse-Seidel-Pre-Jacobi (**VI_GS_J**).

Pour bien observer la différence entre ces algorithmes, nous allons augmenter l'espace d'état, et comparer chaque fois les algorithmes, pour qu'il soit un espace très grand, afin de prouver remarquer la différence.

Dans le chapitre précédent, nous avons clairement remarqué la déférence importante entre ces deux algorithmes, chose qu'est normal, puisque l'algorithme **VIGS**, contient une étape clé, qui sert à faire la mise à jour juste après chaque calcule, ainsi on doit bien attendre une réduction de temps au niveau d'exécution, chose qu'on va le voir au graphe qui vient par la suite :

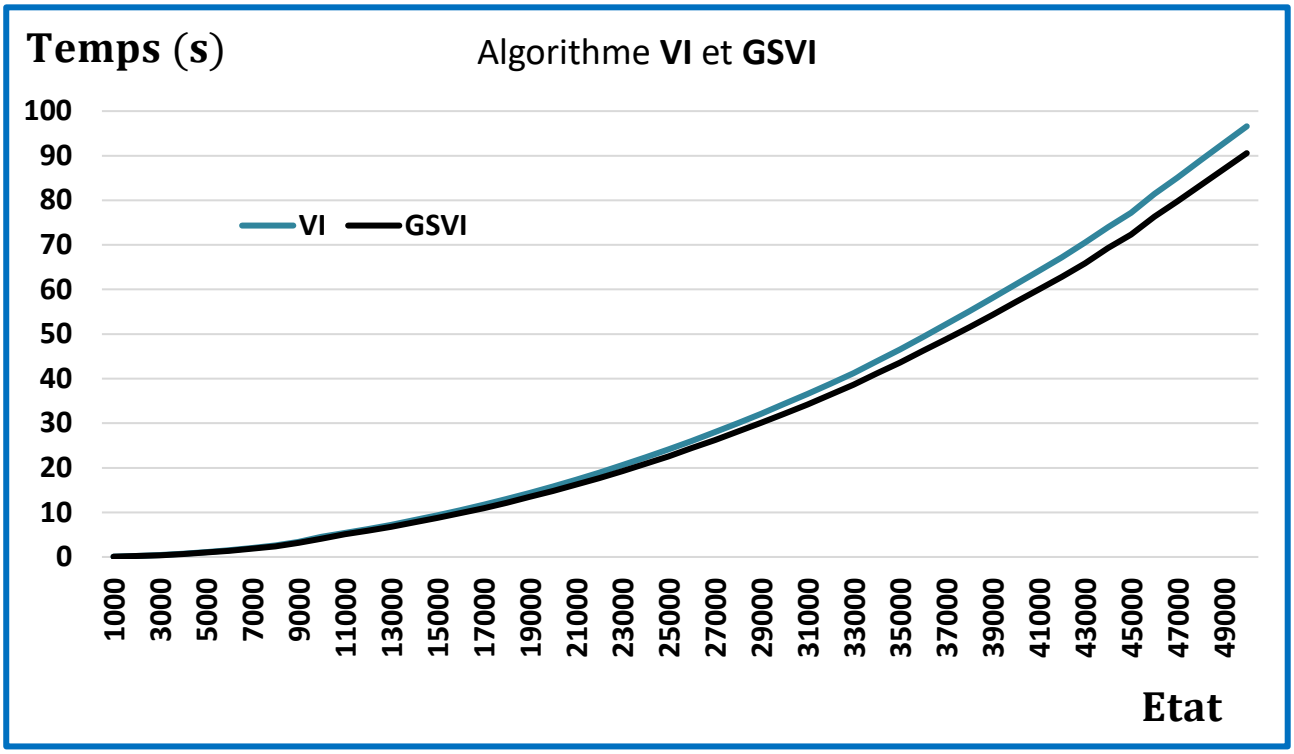

Figure 9 : Comparaison entre l'algorithme d'itération de la valeur et Gauss-Seidel

Notons que la généralisation d'un espace d'état assez grand, nombre de successeurs aléatoires, nombre d'actions (dans chaque états) aussi aléatoire, ainsi des probabilités de transition aléatoire (qui vérifient bien sûr les caractéristiques d'une matrice stochastique, tout ça à nous aider d'avoir ce graphe afin de découvrir la différence (qu'est claire d'après le graphe) au niveau de temps d'exécution, et remarquer clairement la différence entre les algorithmes. D'après le graphe, il est clair que l'algorithme d'itération de la valeur à gagner la compétition au niveau du temps d'exécution, et surtout lorsque le nombre d'état est très grand (plus de **50 000** états), à ce moment on peut voir la différence importante entre les algorithmes.

Notons que dans cet exemple, on a utilisé un nombre d'action aléatoire dans chaque état, ce nombre varie entre 1 et 10, alors qu'est un nombre pas très grand, mais malgré ça on a observé une différence importante. Nous allons encore un peu loin en augmentant le nombre d'action généré en chaque état, afin d'assurer la différence entre ces deux algorithmes.

Le nombre d'action utilisé dans ce cas est toujours aléatoire, mais qu'est trois fois plus que celui utilisé avant, ainsi, et comme nous le remarquons, la différence entre les deux algorithmes n'est plus grande.

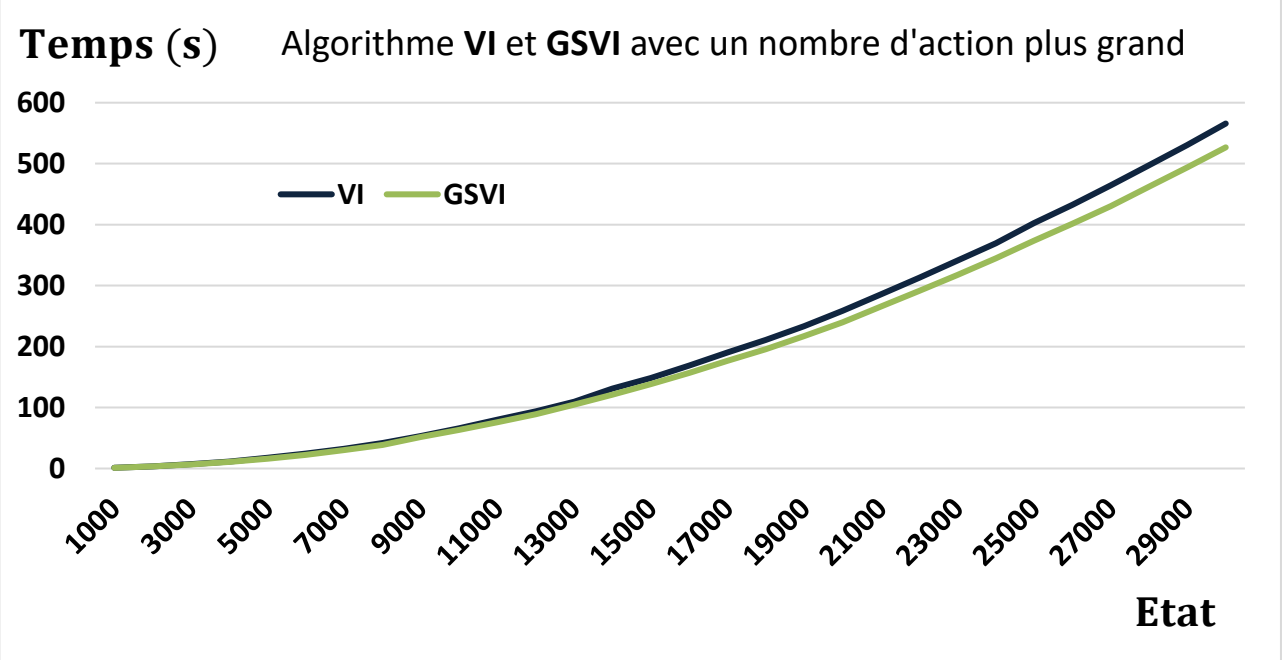

Figure 10 : Comparaison entre l'algorithme d'itération de la valeur et Gauss-Seidel avec un nombre d'action plus grand

Comme nous le remarquons à partir de ce graphe, même avec un nombre d'action encore grand, la différence entre les deux algorithmes (**VI** et **GSVI**) existe toujours, et cette fois elle est devenue importante, alors l'algorithme **GSVI** à prouver sa performance, alors on peut prédire que plus qu'on augmente le nombre d'action, plus qu'on observe une différence encore importante, d'où la compétition est gagnée par **GSVI**.

En générale l'algorithme de l'itération de la valeur modifié, reste caractérisé par sa convergence rapide en le comparant avec l'itération de la valeur, pour voir ça encore plus clairement, nous allons montrer l'évolution de ces deux algorithmes, en montrant la valeur de $\left|\left|\boldsymbol{V^{n-1}} - \boldsymbol{V^{n}}\right|\right|$ en fonction du nombre itération, prisent pour la convergence.

Pour bien observé cela, nous allons essayer de visualiser le graphe, de la valeur, de la norme de la déférence, entre les deux dernières valeurs de vecteurs gains, dans chaque itération, afin de voir clairement que l'algorithme **GSVI**, se rapproche de la convergence, rapidement, par rapport à l'algorithme **VI**, comme le montre le graphe qui vient par la suite :

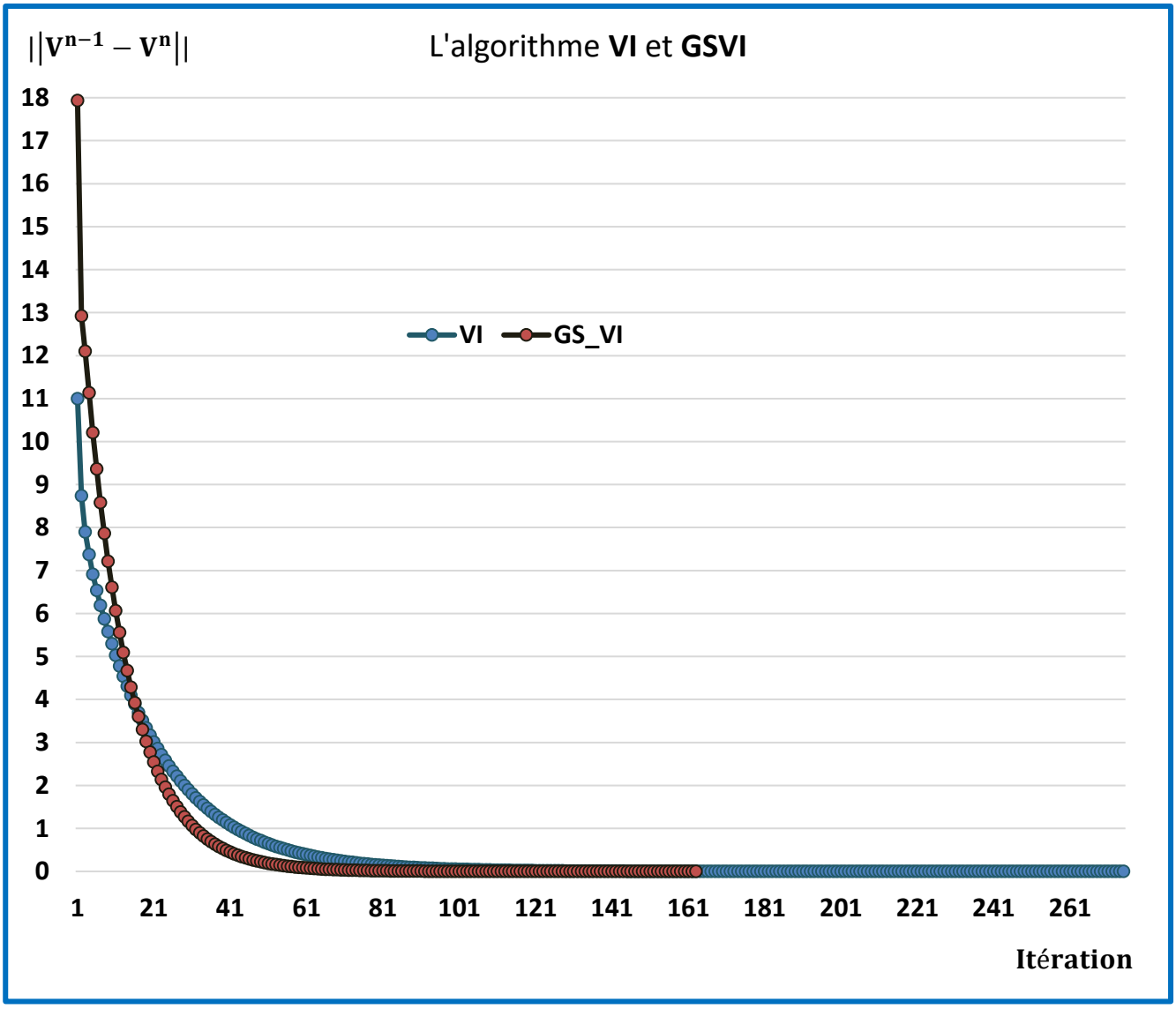

Figure 11 : Comparaison au niveau d'itération entre l'algorithme VI et GSVI

D'après ce graphe, on remarque que lorsque l'algorithme d'itération de la valeur a pris **261** itérations pour converger, l'algorithme **GSVI**, a convergé juste en prenant **161**, **a**insi, on conclut que ce dernier a convergé juste en **61.68%** de nombre d'itération, par rapport à **VI**. Cette différence est due au fait que, dans l'algorithme **GSVI** il y a une mise à jour de chaque valeur calculée du vecteur de gain **V**. Notons que la modification d'un composant dans le ce vecteur, implique une influence sur la convergence de l'algorithme, et puisque cette modification se fait juste après les calculs de chaque composante, cela implique que ce changement va réduire le nombre d'itération. Ce qu'est tout à fait normale, vu que l'algorithme ne va pas attendre jusqu'à la fin de tous les calculs de **V**, pour faire la mise à jour, d'où la diminution de nombre d'itération.

La performance de l'itération de la valeur de **Gauss-Seidel** peut être améliorée en réorganisant l'ordre suivant, avant de calculer (équation de Bellman), à l'étape **2.5** de l'algorithme, ou d'inverser la direction de la ligne ou en inversant la direction de l'axe évaluation des états à chaque passage dans l'algorithme.

Alors, une autre amélioration vient par la suit, qu'est autre version de cet algorithme, s'appelle **Pre-Jacobi**, qui consiste à décomposer la matrice de transition, comme somme de deux matrices, ou l'une de cette matrice est diagonal. Sachant qu'une matrice inversible est facile à calculer, d'où la

performance de cet algorithme. En appliquant l'algorithme Itération de la valeur Jacobi (**VI_J**), on constate qu'il prend moins du temps que **VI** pour atteindre la convergence. Notons que d'autres approches peuvent être appliquées, par exemple : La combinaison de **Jacobi** et **GS.** Une comparaison sera présentée juste après, de l'algorithme **VI, VI_J, GS** et **GS_J**. pour pouvoir enfin, distinguer la différence entre l'algorithme **VI** et toutes ses variantes.

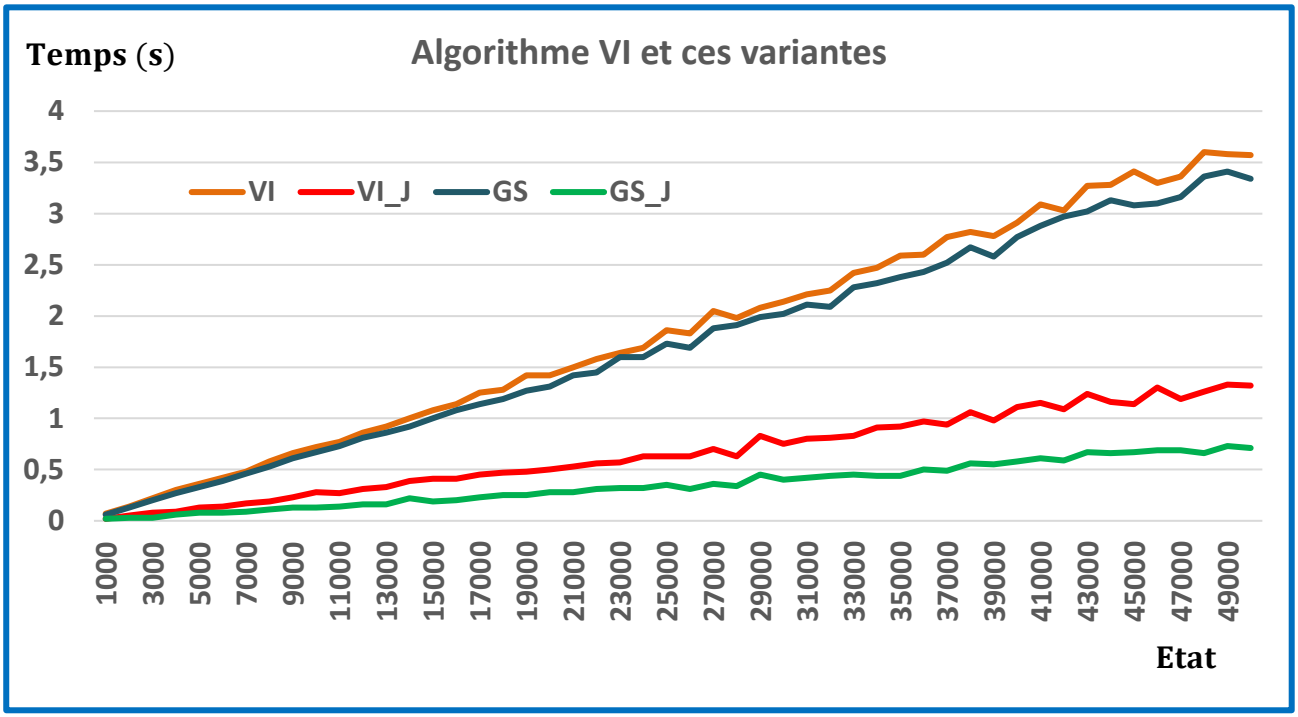

Figure 12 : Comparaison entre les algorithmes : VI, GS, VI_J et GS_J

Après avoir comparé l'algorithme **VI** avec ses variantes (**GS, VI_J** et **GSVI**_J), nous avons constaté que l'algorithme **VI**, et malgré sa simplicité au niveau de programmation, qui revient à la résolution itérative du système linéaire, les variantes **GSVI, VI_J** et **GSVI_J**, avaient encore accélérer cet algorithme, soit au niveau de temps d'exécution et même au niveau d'itération nécessaires pour converger.

Il y a encore beaucoup des paramètres, qui peuvent influencer la convergence de l'algorithme ou même le nombre d'itération, pour la convergence de l'algorithme, parmi ces paramètres, il y a **le facteur d'actualisation** $\propto$. Comme nous l'avons vu aux chapitres précédents, ce paramètre, qui sert à actualiser la valeur des récompenses, afin de les espérer, pour calculer les vecteurs gains optimaux, varie entre **0** et **1** strictement. Pour étudier l'influence de ce paramètre sur la convergence, on va faire varier la valeur de ce paramètre, sur un espace d'état toujours aléatoire et très grand ($\mathbf{5 \times 10^5}$), puis on va voir le temps prise pour chaque algorithme, pour atteindre la convergence. Nous allons alors tracer le temps d'exécution en fonction de la valeur de $\propto$

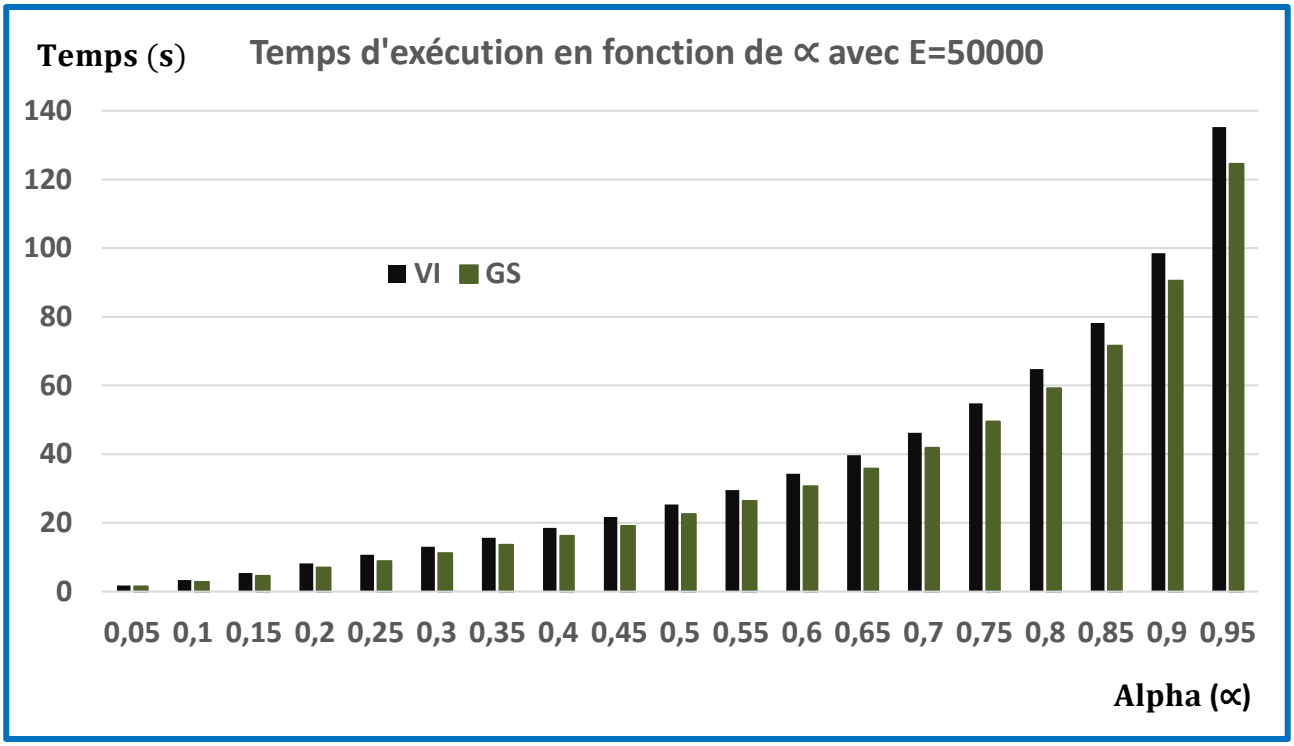

Figure 13 : Etude de temps de convergence de l'algorithme VI selon Alpha sur un espace d'état E=50.000

Il est clair d'après ce graphe, que le choix du paramètre d'actualisation ∝, influence la convergence de l'algorithme, alors, tant qu'on augment ce paramètre, tant que le temps d'exécution s'augmente. Il nous reste maintenant qu'observer l'évolution de nombre d'itération en fonction toujours de ce paramètre. Nous allons ainsi, faire le graphe de nombre d'itération en fonction de ce paramètre ∝, comme le suivant :

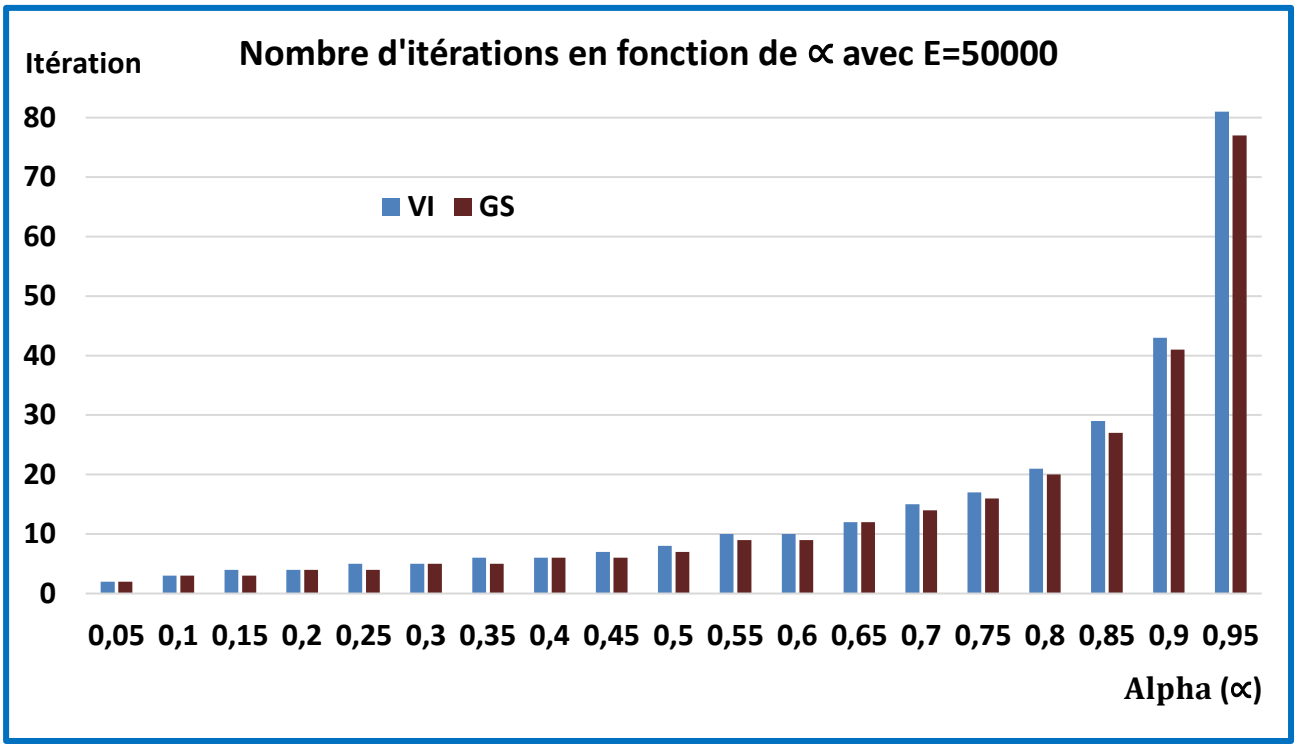

Figure 14 : Etude de convergence selon le temps en changeant Alpha, sur un espace E=50.000

3.3. Algorithme d'itération de la politique et ses variantes

L'algorithme d'itération des politiques (**PI**), est une autre méthode de résolution des problèmes des problèmes de décision Markoviens à horizon infini. Bien que l'itération des valeurs puisse être considérée comme une application d'une approche générale pour trouver des points fixes, l'itération des politiques apparaît directement liée à la structure particulière des problèmes de décision de Markov. Elle s'applique aux problèmes stationnaires d'horizon infini, mais ce n'est pas un algorithme efficace pour résoudre les problèmes d'horizon fini.

L'algorithme **PI** possède une forte complexité, et cela reviens la résolution directe du système linéaire premièrement, ainsi que la parcourt de tous les états ainsi que toutes les actions, sans prendre en compte le nombre d'itération, puisqu'il est en générale pas grand dans cet algorithme, chose qu'on va le voir par la suite dans la version modifier de cet algorithme. Notons que dans l'algorithme d'itération de la politique, la résolution directe du système linéaire est de complexité polynomiale d'ordre 3 qu'est $\mathbf{S^3}$, ainsi que le parcourt de tous les états et toutes actions est de $\mathbf{S} \times \mathbf{A}$, par la suite la complexité de cet algorithme, qu'est $\mathbf{o}(\mathbf{S^3} + \mathbf{S} \times \mathbf{A})$, d'où la difficulté de l'implémentation de cet algorithme.

De même que l'algorithme **VI**, il y a une version modifiée de cet algorithme (**MPI**), ou, à la place d'une résolution directe de l'algorithme, il y a une résolution itérative, ainsi, une complexité d'ordre **3** avec la parcourt des actions, alors la complexité devienne : $\mathbf{o}(\mathbf{S^3} \times \mathbf{A})$. Cette version, garde le nombre d'itération, qui prend l'algorithme pour converger, et ce nombre est toujours très petit. Le tableau qui vient, montre l'évolution du vecteur gain optimal **V**, ainsi que la stratégie optimale nommé **Sopt**. L'algorithme est arrivé d'atteindre la convergence en **2** itérations, avec les valeurs de **V** et **Sopt**, qui sont les suivants :

Tableau 3 : Les valeurs de vecteur gain V et stratégies, optimales dans chaque itération, de l'algorithme MPI

Itération 1	
Sopt[1]=0	Vopt[1]=0.000000
Sopt[2]=0	Vopt[2]=0.000000
Sopt[3]=0	Vopt[3]=0.000000
Sopt[4]=0	Vopt[4]=0.000000
Sopt[5]=0	Vopt[5]=0.000000
Sopt[6]=0	Vopt[6]=0.000000
Sopt[7]=0	Vopt[7]=0.000000
Sopt[8]=0	Vopt[8]=0.000000
Sopt[9]=0	Vopt[9]=0.000000
Sopt[10]=0	Vopt[10]=0.000000
Sopt[11]=0	Vopt[11]=0.000000
Sopt[12]=0	Vopt[12]=0.000000
Sopt[13]=0	Vopt[13]=0.000000
Sopt[14]=0	Vopt[14]=0.000000
Sopt[15]=0	Vopt[15]=0.000000
Sopt[16]=0	Vopt[16]=0.000000
Sopt[17]=0	Vopt[17]=0.000000
Sopt[18]=0	Vopt[18]=0.000000
Sopt[19]=0	Vopt[19]=0.000000
Sopt[20]=0	Vopt[20]=0.000000

Itération 2	
Sopt[1]=0	Vopt[1]=124.728821
Sopt[2]=5	Vopt[2]=97.469635
Sopt[3]=6	Vopt[3]=102.113617
Sopt[4]=0	Vopt[4]=171.843567
Sopt[5]=4	Vopt[5]=154.376038
Sopt[6]=11	Vopt[6]=55.843632
Sopt[7]=1	Vopt[7]=124.728821
Sopt[8]=0	Vopt[8]=211.238190
Sopt[9]=4	Vopt[9]=107.255905
Sopt[10]=3	Vopt[10]=121.566559
Sopt[11]=3	Vopt[11]=115.566559
Sopt[12]=1	Vopt[12]=48.469654
Sopt[13]=2	Vopt[13]=111.469620
Sopt[14]=7	Vopt[14]=106.824249
Sopt[15]=0	Vopt[15]=135.238205
Sopt[16]=3	Vopt[16]=160.255875
Sopt[17]=2	Vopt[17]=104.843613
Sopt[18]=0	Vopt[18]=153.854599
Sopt[19]=0	Vopt[19]=139.244080
Sopt[20]=2	Vopt[20]=102.244110

Nous confirmons maintenant que l'algorithme **MPI** ne fait pas beaucoup d'actions pour atteindre la convergence, mais comme nous l'avons indiqué juste avant, le traitement qu'est dans chaque itération est le plus couteux, pour ça encore plus clairement nous allons varier l'espace d'états et calculer le temps d'exécution, afin d'avoir un graphe qui lie ces deux paramètres, comme le suivant :

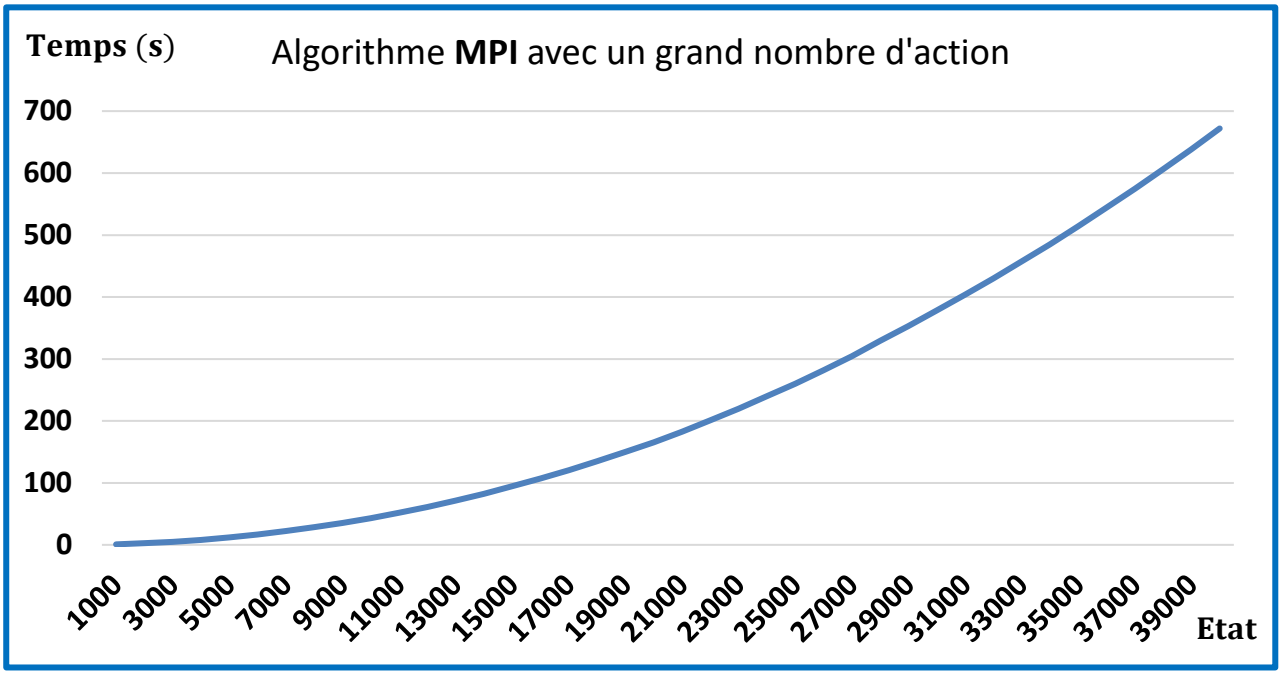

Figure 15 : Evolution selon le temps en fonction d'espace d'état, de l'algorithme PI

Notons que nous avons appliqué l'algorithme **MPI** sur des espaces d'états assez grands, ainsi que les nombre d'actions, qu'est aléatoire, et dépasse **9000** actions par état. Nous allons voir ça encore plus détaillé dans la section suivante, en faisant une comparaison de cet algorithme avec la version modifié de l'algorithme **VI**, avec des grands nombres d'actions.

L'algorithme d'itération de la politique, commence avec une initialisation d'une politique, pour commencer le traitement, afin de converger lorsque cette politique ne se change plus. En général cette politique s'initialise par la même action pour toute cette politique. Nous allons maintenant essayer de faire une comparaison entre deux types d'initialisation, dans laquelle on va initialiser une fois, de la façon classique (la même action dans toute la politique) est l'autre fois, on initialise la politique aléatoirement, afin de voir la différence entre ces deux méthodes. Pour exhiber la différence, nous allons calculer le temps d'exécution nécessaire pour la convergence pour chacune de ces deux méthodes comme le montre le graphe suivant :

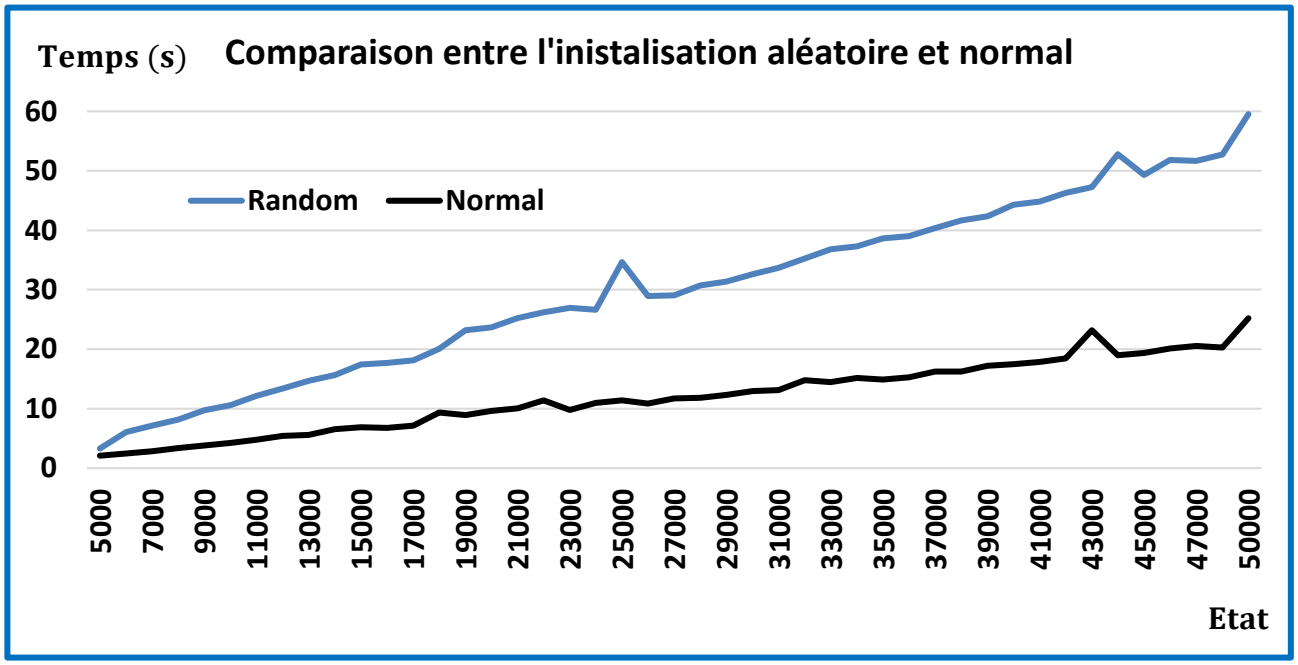

Figure 16 : Comparaison entre l'initialisation aléatoire et normal de la politique, avec l'algorithme MPI

3.4. Algorithmes d'itération de la valeur et d'itération de la politique

Comme nous l'avons signalé, l'algorithme de l'itération de la politique, et vu à sa forte complexité, qui revient à la résolution directe du système d'équation. Notons que la faiblesse de cet algorithme au niveau de temps d'exécution, n'es pas due au grand nombre d'itération, par contre, et comme nous l'avons juste avant, l'algorithme **PI**, ne prendre pas beaucoup d'itération pour converger, mais ce qu'il prend beaucoup de temps, c'est que dans chaque itération, il y a une résolution directe d'un système linéaire. Nous allons premier temps comparer les deux algorithmes suivants : **VI** et **PI**, puis, et comme nous avons vu au niveau de l'algorithme **VI**, après avoir modifié cet algorithme (chose fait en **GSVI**), alors on a obtenu une réduction importante au niveau de temps d'exécution. Alors on va faire la même chose avec l'algorithme **PI**, afin d'avoir une version

modifier de cette algorithme nommé **Gauss-Seidel Policy Itération** (**GSPI**), qu'on va la voir par la suite, pour la comparer après avec la version modifier de l'algorithme **VI**.

Tout d'abord, est pour bien observer la déférence d'efficacité entre l'algorithme **VI** et **MPI,** une comparaison est faite entre l'algorithme **VI** et **MPI**, au niveau de temps d'exécution, sur des espaces d'état assez grands, comme le montre le graphe ci-après, puis on va comparer les deux algorithmes modifiés.

Comme le montre le graphe ci avant, la déférence entre l'algorithme **PI** et **VI** est claire, chose qui était attendu d'abord, et cela revient à la résolution directe du system d'équation, bien que l'algorithme **MPI** ne fait pas beaucoup d'itération, mais la forte complexité dans chacune est trop élevée par rapport à l'algorithme **VI**.

Nous allons passer maintenant aux versions modifier de ces deux algorithmes, afin de les comparer, sur les mêmes espaces d'état qu'avant.

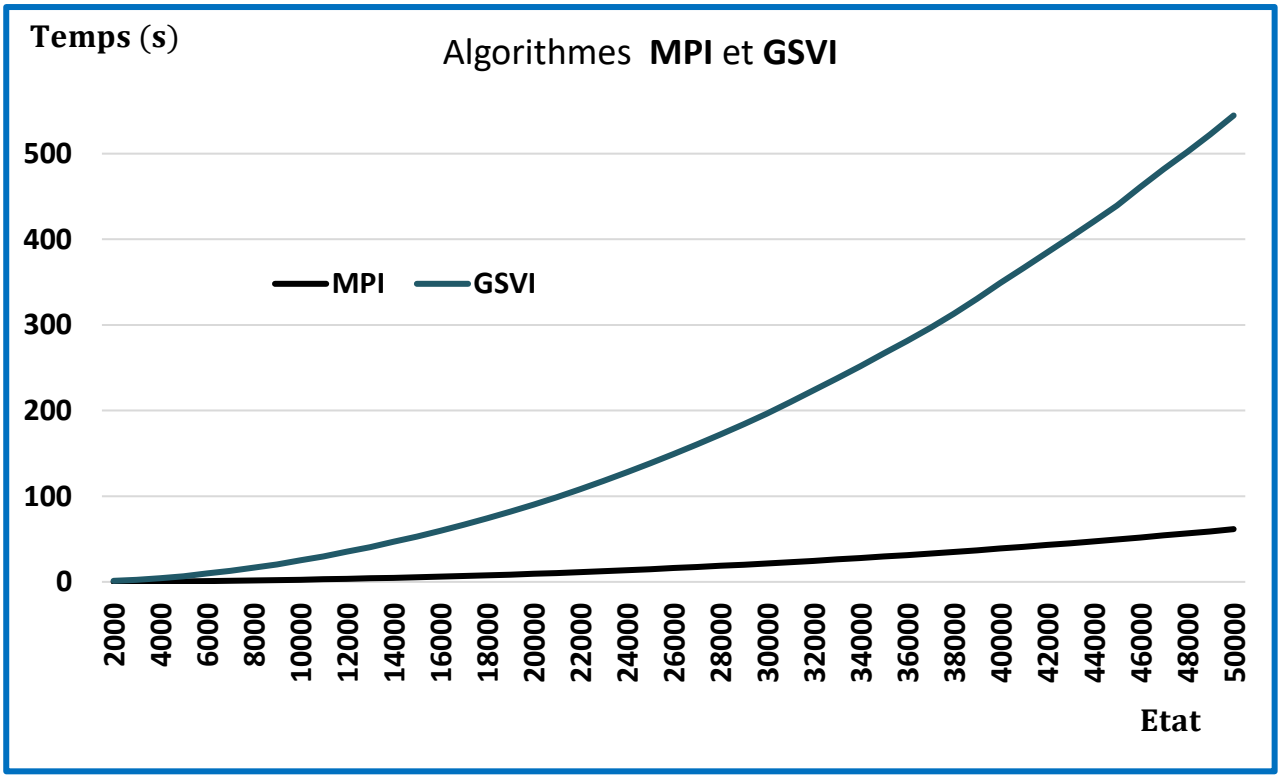

Figure 17 : Comparaison au niveau de temps d'exécution de l'algorithme MPI et GSVI selon le nombre d'états

3.5. Algorithmes parallèles

L'espace d'états des **PDM**, s'augmente d'une façon exponentielle, ce qui implique une augmentation soit au niveau de temps d'exécution ou de la mémoire. Puterman dans [1] a proposé une procédure d'élimination des actions, afin de réduire la complexité de l'algorithme et le temps d'exécution.

Malheureusement les **PDM** de grandes tailles semblent d'être très compliqué à les résoudre, et consomment beaucoup du temps et beaucoup d'espace mémoire.

Ils existent beaucoup d'études sur l'utilisation de parallélisme au **PDM**, dans **[3]** JIRÍ BARNAT et JAKUB CHALOUPKA, ont proposé une approche pour la détection des Composantes Fortement Connexes (**CFC**), avec un algorithme parallèle.

Dans ce chapitre nous allons proposer une version de l'algorithme de l'itération de la valeur, qui sera un algorithme parallèle, pour la réduction du temps d'exécution.

3.5.1. Le parallélisme

La **programmation parallèle** consiste à exécuter plusieurs tâches en même temps ! Un jardinier, par exemple peut arroser ses Plantes et répondre en criant oui, mais le jardinier doit quand même faire deux actions en même temps : arroser et crier. Cela est possible, car l'homme est une machine formidable capable de réaliser de très nombreuses choses en même temps.

Cependant pour un ordinateur, ce n'est pas aussi simple. Car initialement, les ordinateurs sont conçus pour traiter les informations séquentiellement : le processeur exécute les instructions les unes à la suite des autres. Dans ce contexte, il est difficile d'imaginer de pouvoir réaliser plusieurs actions en même temps. Pourtant, quand on utilise nos ordinateurs et que nous copion un (gros) fichier, on peut toujours continuer à naviguer dans nos répertoires.

On n'est pas (plus) oblige d'attendre la fin de la copie. Ceci est possible, car le système d'exploitation gère ce qu'on appelle un **Schedule** (ordonnanceur en français). Il va allouer un certain temps d'exécution à chaque code qui désire fonctionner à cet instant. Ainsi, il va exécuter un premier code (comme la copie du fichier), puis l'interrompre pour démarrer un deuxième code (comme la navigation dans les répertoires), puis interrompre ce dernier pour relancer le premier là où il en était précédemment ... et alterner ainsi de suite avec tous les programmes qui tournent à ce moment. Bien que chaque code s'exécute à tour de rôle, il s'agit bien d'actions simultanées puisque lorsque la première tâche est en cours d'exécution, la deuxième est également en cours d'exécution.

Depuis les années **2000**, les ordinateurs multiprocesseurs ou les processeurs multi cœurs se sont généralisés. Aujourd'hui, pratiquement tout le monde possède un ordinateur disposant de cette technologie. Or cette technologie permet de réellement exécuter plusieurs instructions en même temps (1 par cœur ou par processeur).

Cependant, pour qu'un programme puisse bénéficier de ces avantages, et donc exécuter deux codes (ou plus) de manière concurrente, il doit indiquer qu'un certain code ne s'exécute plus dans la même pile que le code principal. Une pile, ou plutôt **pile sémantique** (pour ne pas la confondre avec la pile d'exécution qui liste les fonctions actives du programme, et même si dans chaque pile sémantique se trouve une pile d'exécution) est un ensemble d'instructions que l'ordinateur doit exécuter. Il est possible que vous entendiez les termes **thread** ou **processus**.

Depuis le début de cette étude, nous avons vu plusieurs fois, le vecteur gains optimal **V**, les méthodes et les composantes de son calcul (pour revoir tout ça il suffit de revenir à l'équation d'optimalité au chapitre **I**).

Notons que pour les calculs de de l'équation d'optimalité $\mathbf{V^n}$, son itératifs et indépendants, alors il faut profiter de cette indépendance, afin de paralléliser les calculs, et les partager entre plusieurs processeur (**Thread**), ou dans chacun on va indiquer le début et la fin des calculs, qui sont

tout simplement l'état du départ et l'état final, autrement dans chaque **Thread** on va calculer les valeurs $\mathbf{V^s}$ jusqu'à $\mathbf{V^p}$, ou les valeurs de **s** et **p** sont fournis au **Thread**. Nous allons alors proposer une version d'un algorithme parallèle d'itération de la valeur.

3.5.2. Algorithme parallèle d'itération de la valeur

Dans cette session nous allons montrer l'algorithme parallèle d'itération de la valeur, alors l'algorithme est présenté comme le suivant :

Algorithme 8 : Itération de la valeur (Algorithme parallèle)

Entrées : $\mathbf{S, A, P, r, \alpha, \varepsilon, N_p}$ // où N_p est le nombre des processeurs

1. **Diviser S en $\mathbf{N_p}$ partitions : $\mathbf{S_0, S_1, \dots, S_{N_p}}$**
   Initialiser : $\mathbf{V_0 = 0}$
2. **Répéter**
   Pour toute partition $\mathbf{S_k, k = 0}$ jusqu'a $\mathbf{N_p}$Faire en parallèle

$$V_{n+1}(s) = max_{a \in A}\left[r(s,a) + \alpha \sum_{S' \in \delta} P(s',a,s)\, V_n(s')\right] \qquad \textbf{3.5.1}$$

   Jusqu'à : $\left|\left|V_{n+1} - V_n\right|\right| < \varepsilon$
3. **Pour toute partition $\mathbf{N_{p_{k'}}, k = 0}$ jusqu'a $\mathbf{N_p}$Faire en parallèle**

$$\pi^*(s) = argmax_{a \in A}\left[r(s,a) + \alpha \sum_{S' \in N_{p_{k'}}} P(s',a,s)\, V_n(s')\right] \qquad \textbf{3.5.2}$$

4. **Retourne π^***

Figure 18 : Algorithme d'itération de la valeur parallèle

Comme le montre l'algorithme parallèle d'itération de la valeur, on a utilisé le parallélisme au niveau de la boucle responsable des calculs du vecteur gains, ainsi que la boucle qui cherche la politique qui maximise la quantité de l'équation **3.5.2**, donc on a parallélisé la recherche de la politique optimale dans chaque état. Ils nous restent maintenant que comparer cet algorithme parallèle, avec l'algorithme séquentiel, afin d'observer la déférence entre les deux, et remarquait la puissance du parallélisme, comme le montre le graphe suivant :

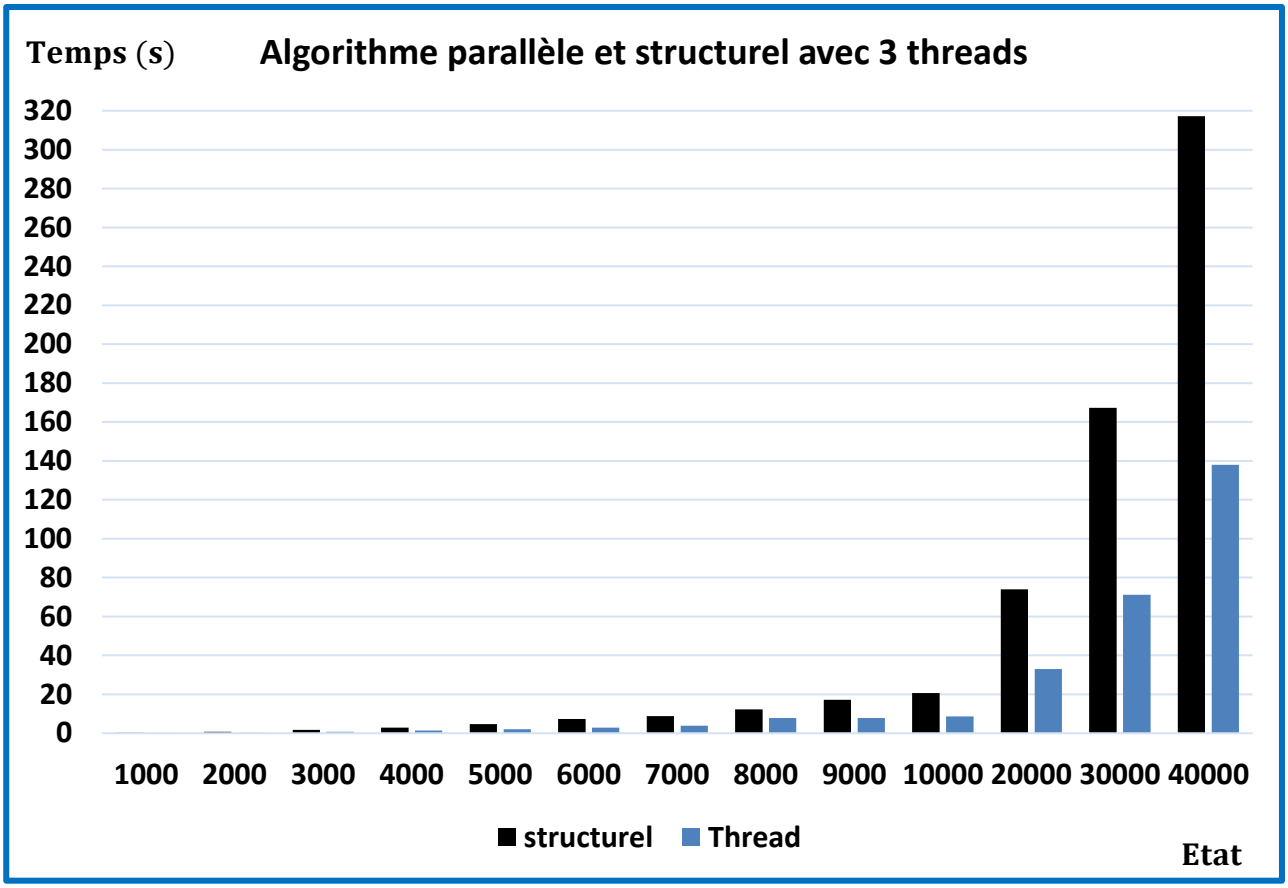

Figure 19 : Comparaison selon le temps d'exécution de l'algorithme VI structurel et VI parallèle

Comme nous constatons d'après le graphe précédent, l'algorithme Parallèle d'itération de la valeur (**PVI**), a prouvé sa performance, avec une distribution entre **3** processeurs (Thread), afin d'avoir ces résultats, ainsi, on peut faire le parallélisme avec plus de processeurs, par exemple **3** ou **4** etc…

Malgré la puissance du parallélisme au niveau de réduction de temps d'exécution, il ne faut pas oublier toute déclaration, création, allocation… coute du temps pour la faire. Le lancement des processeurs, et leurs appels… coûtent aussi du temps, ce qui implique, en augmentant la division des calculs par beaucoup des Threads, on aura ainsi, un grand nombre des valeurs échangés, beaucoup de thread à déclarer, et surtout beaucoup d'appellation au niveau de ces threads. En bref, alors il y'aurai beaucoup de temps fournit juste pour la déclaration, appellation et surtout le lancement de ces Thread, et aussi l'échange des données entre eux.

Cela implique, qu'une autre étude est nécessaire au niveau de nombre de Threads, en variant ce nombre et observer le temps d'exécution, afin de savoir l'influence de nombres sur le temps *d'exécution. Pour cela nous avons essayé de tracer la fonction suivante : $\mathbf{f(n) = t}$, ou $\mathbf{n}$ est le nombre de processeurs, et $\mathbf{t}$ est le temps d'exécution. Alors, en variant le nombre de processeurs, avec lesquels on va diviser les calculs, on va voir le temps pris pour que le programme termine les calculs.

Les résultats obtenus, sont montrés sur le graphe ci-après :

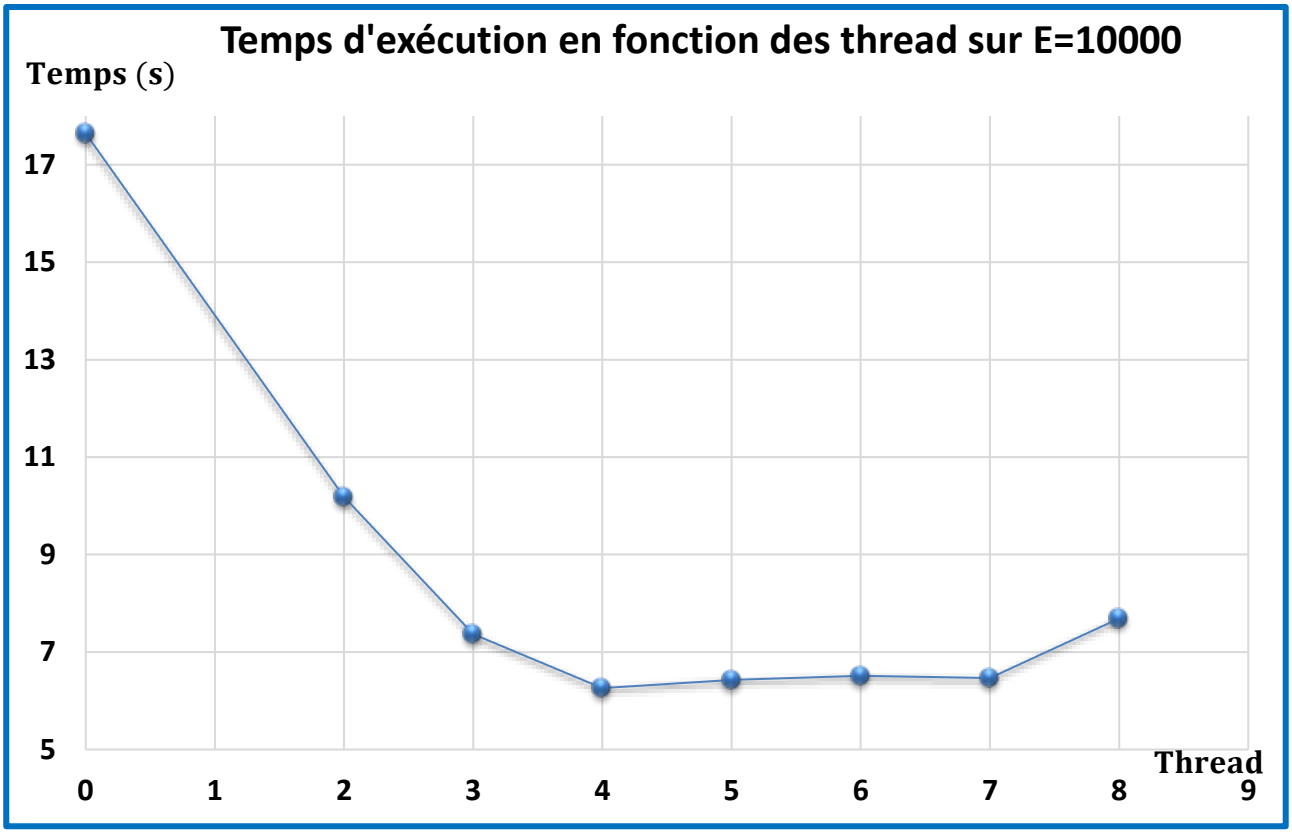

Figure 20 : Etude d'évolution du temps d'exécution suivant le nombre de threads

Nous avons commencé avec un nombre de threads, qui vaut 2, puis on a augmenté ce nombre afin d'observé l'influence de ce nombre sur le temps d'exécution.

Notons qu'à partir d'un nombre de threads utilisés, on ne gagne plus du temps, et plus précisément, lorsqu'on dépasse le nombre des processeurs de la machine, qu'est 3 processeurs, dans notre cas. Ainsi que, si on continu l'augmentation du nombre de threads, le temps s'augment, et cela, en fait reviens à temps nécessaire du démarrage, déclaration des threads, attentes de la fin du dernier thread, et surtout l'échange des données entre les threads, alors à partir à certain nombre, on peut dire qu'on perde plus qu'on gagne, d'où l'augmentation du temps d'exécution. Nous allons passer maintenant, dans la section qui vient par la suite, à la version parallèle de l'algorithme **MPI**, afin de la comparer avec la version structurelle, puis passer à la comparaison entre les deux algorithmes parallèles, pour savoir qui va gagner la compétition.

3.5.3. Algorithme parallèle d'itération de la politique modifié

Algorithme 9 : Algorithme parallèle d'Itération de la politique modifié

Entrées : $\mathbf{S, A, P, r, \alpha, \varepsilon, N_p}$ // où N_p est le nombre des processeurs

1. **Diviser S en $\mathbf{N_p}$ partitions : $\mathbf{S_0, S_1, \dots, S_{N_p}}$**

t=0

Initialiser $\boldsymbol{\pi^0}$

Répéter

Initialiser : $\mathbf{V_0 = 0}$

n = 0

Répéter

Pour toute partition $\mathbf{S_k, k = 0}$ jusqu'a $\mathbf{N_p}$Faire en parallèle

$$V_{n+1}(s) = \left[r(s, \pi^0(s)) + \alpha \sum_{S' \in \delta} P(s', \pi^0(s), s)\, V_n(s') \right] \qquad 3.5.1$$

n = n+1

Jusqu'à : $\left|\left|V_{n+1} - V_n\right|\right| < \varepsilon$

t = t+1

Pour toute partition $\mathbf{S_k, k = 0}$ jusqu'a $\mathbf{N_p}$Faire en parallèle

$$\pi^t(s) = Argmax_{a \in A} \left[r(s, a) + \alpha \sum_{S' \in \delta} P(s', a, s)\, V_n(s') \right] \qquad 3.5.1$$

Jusqu'à : $\boldsymbol{\pi^{t-1} = \pi^t}$

2. **Retourne $\boldsymbol{\pi^*}$**

Figure 21 : Algorithme d'itération de la politique parallèle

Comme le montre l'algorithme montré ci-dessus, le parallélisme était fait, toujours, au niveau des calculs du vecteur gains, ainsi que la recherche de la politique optimale, vu qu'il y a toujours l'indépendance des calculs pour ces calculs.

Après avoir montré l'algorithme parallèle, et comme nous avons fait avec l'autre algorithme parallèle, nous allons comparer l'algorithme d'itération de la politique modifié parallèle, avec l'algorithme séquentiel, afin de voir la déférence au niveau de temps d'exécution, et prouver son efficacité, nous allons le comparer l'algorithme structurel, au niveau de temps d'exécution, comme le montre le graphe suivant :

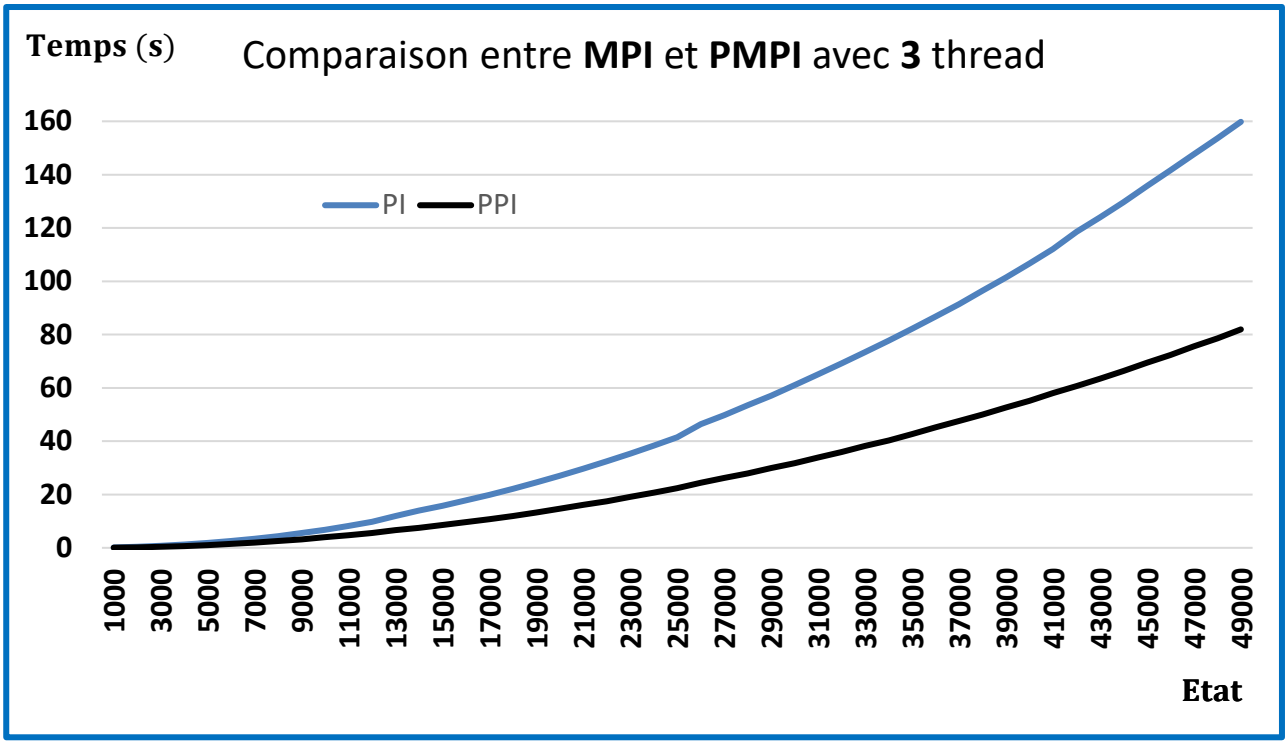

Figure 22 : Comparaison entre les algorithmes d'itération de la politique structurel et parallèle

Comme nous l'avons avisé (normalement), voilà, l'algorithme parallèle d'itération de la politique **PMPI** qui prend beaucoup moins de temps par rapport à l'algorithme structurel **MPI**, notons, que le nombre de threads utilisés ici est toujours trois threads, et cela était suffisant pour avoir une réduction importante au niveau de temps d'exécution qui vaut presque **50%**. Il nous reste maintenant qu'à faire une comparaison entre les algorithmes parallèles, chose qu'on va faire dans la section suivante.

3.5.4. Comparaison des algorithmes parallèles

L'objectif de cette section, est de comparer les deux versions parallèles des algorithmes **PVI** et **PMPI**, qu'on déjà réaliser, afin de sortir avec un résultat, qui nous dit qui restera le meilleur entre ces deux versions parallèles de ces algorithmes, afin de juger qu'elle la version parallèle la plus performante. On peut prévenir le résultat qu'on va obtenir, vu que l'algorithme **MPI** était très rapide au niveau de temps d'exécution, par rapport aux autres algorithmes, alors normalement, et même pour les versions parallèles, cet algorithmes (**MPI**) doit être plus rapide que (**VI**), chose qu'on va le voir en comparant ces deux algorithmes dans le même graphe, selon le temps d'exécution. Pour se faire nous allons évoluer le nombre d'états, toujours avec un nombre d'action aléatoire, en observant le temps d'exécution, comme le montre le graphe qui par la suite.

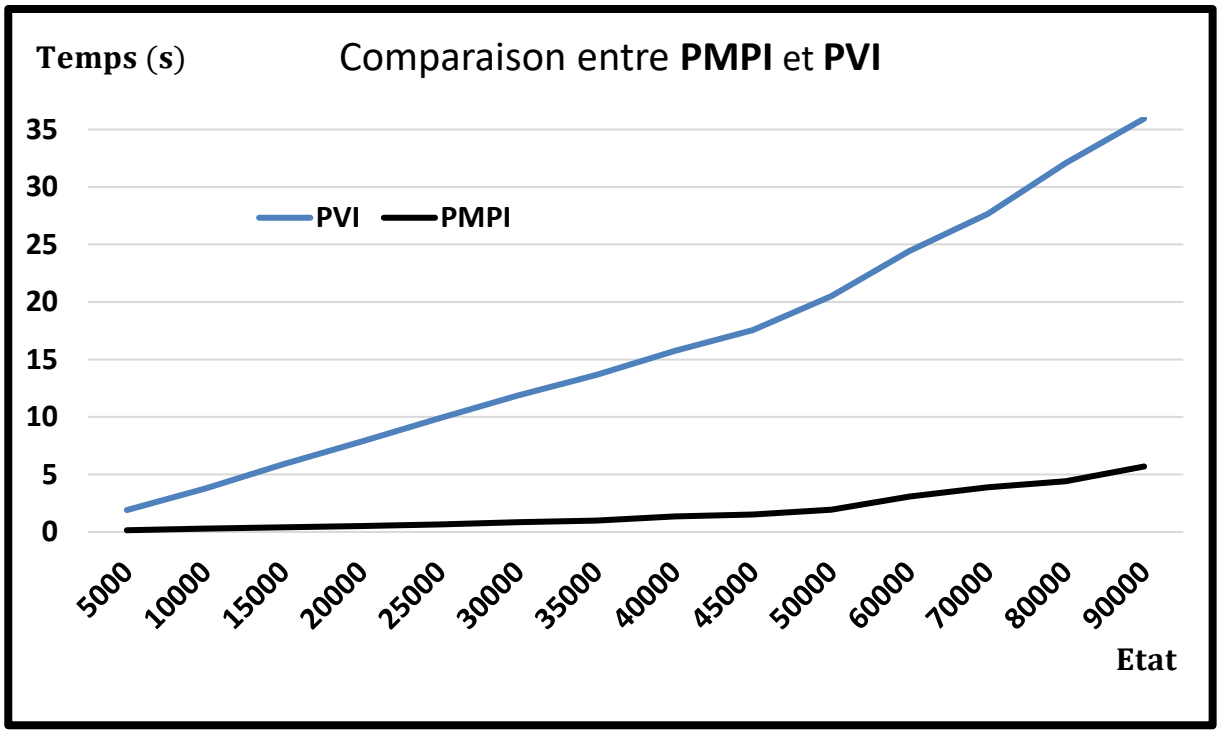

Figure 23 : Comparaison entre les algorithme parallèles, itération de la valeur et l'itération de la politique modifié

2.6. Conclusion

Les deux algorithmes PVI et **PMPI** ont prouvé leurs efficacités au niveau de temps d'exécution, suite à la réduction importante au niveau de temps de calcul, par rapport aux algorithmes séquentiels. Mais, la comparaison entre les algorithmes nous montre clairement que l'algorithme **PVI** est plus rapide par rapport à l'algorithme **PMPI.** Une contrainte très importante qu'il faut la prendre en considération, qu'est le nombre des threads utilisés pour résoudre un **PDM**, par exemple il ne sert à rien d'appliquer **50** Processeurs, sur un espace de **100** états, vu qu'on commencera à ce niveau de perdre plus qu'on gagne. Alors il faut prendre bien cette distribution en considération, qu'est connue par **la distribution statique.**

Dans le travail qu'on a effectué jusqu'à maintenant, on a présenté essayer de profiter de l'indépendance entre les calculs, pour faire des versions parallèles des algorithmes étudiés. Nous allons maintenant, essayé des ces algorithmes, les méthodes de décompositions aux niveaux, ainsi que des calculs mathématiques, afin de sortir avec un algorithme efficace, à appliquer sur des **PDM** de grandes tailles. Vu qu'un gros **PDM**, peut être décomposé en sous-**PDM**, alors il faut concevoir une nouvelle méthode efficace, afin de savoir interagir avec la variation de la taille de chaque sous-**PDM** (ou encore **CFC**), et de la combiner avec l'algorithme **PVI**, pour augmenter ainsi la performance au niveau de la résolution des **PDM** de grande taille. Cette nouvelle approche sera le but du prochain chapitre, ainsi on va l'appeler **Affectation Dynamique des processeurs**.

Chapitre V : Affectation dynamique des processeurs

4.1. Introduction

Dans les chapitres précédents nous avons étudié les solveurs **PDM**, les comparer, et nous avons proposé des versions parallèles des algorithmes **VI** et **MPI**, afin de pouvoir réduire le temps d'exécution. Ces deux versions, d'algorithmes ont prouvé sa puissance après avoir les comparé à d'autres algorithmes classiques. Maintenant, et dans ce chapitre, nous allons passer à une notion très importante aussi, qu'est la décomposition en niveau et la détection des composantes fortement connexes (**CFC**). Notons que cette notion a une forte liaison avec, l'utilisation du parallélisme, en effet, la décomposition en niveau et la détection des **CFC**, permet de subdiviser l'espace d'état, en sous espace, chacun dans un niveau, ou chaque contient que des **CFCs**.

Beaucoup d'études se concentraient sur la détection des **CFC** et la décomposition en niveau. Vera Matei dans **[10]**, a fait une comparaison entre les algorithmes existe, pour la décomposition en niveau, pour sortir avec une conclusion, sur les plus performants. L'étude faite en **[3]** et Vincent Bloemen dans **[5]**, ont proposé un algorithme parallèle pour la détection des **CFC**. Mais tous ces études, ne prends pas en considération la différence importante des tailles, dans chaque composante fortement connexe, en chaque niveau, ce qu'il peut être une cause qui ralentir les calculs parfois, par exemple si on deux **CFC**, ou la taille de la première et dix-fois grand que la deuxième, alors ça ne sera pas efficace de distribuer les processeurs, d'une façon égale pour les calculs de ces deux **CFC**. Nous allons profiter des études, et aussi d'algorithme existantes, afin de pourrais les combiner avec notre version parallèle de l'algorithme d'itération de la valeur (**PVI**), pour sortir avec une nouvelle approche. Les résultats finaux, seront implémentés sous forme des graphes, pour montrer l'efficacité de cette approche, par rapport à l'approche standard.

4.2. Composantes fortement connexes

Avec l'augmentation des tailles des **PDM**, la nécessité des méthodes, approches ou outils est devenue plus en plus importantes, pour aider à subdiviser les grandes espaces d'état, à sous espace ou chacune est appelé composante fortement connexes (**CFC**). Une composante fortement connexe est un sous graphe, qu'on peut le considérer aussi comme un petit **PDM**, ou il y a : $\forall\, \mathbf{i}, \mathbf{j} \in \mathbf{G} : \mathbf{P}(\mathbf{i}, \mathbf{j}) > \mathbf{0}$. Autrement dit, entre n'importe quels deux nœuds, il existe un arc qui les lie.

4.3. Décomposition en niveaux

En cherchant toujours dans le sens de réduction de la taille des **PDM**, on trouve la décomposition en niveau des **PDM**, qui sert à décomposer l'espace d'état, en sous niveau $(\mathbf{L_1}, \mathbf{L_2}, \dots, \mathbf{L_{N-1}}, \mathbf{L_N})$, ou chaque niveau contient une ou plusieurs **CFC**. La figure suivante, montre un exemple d'un graphe décomposé en niveau :

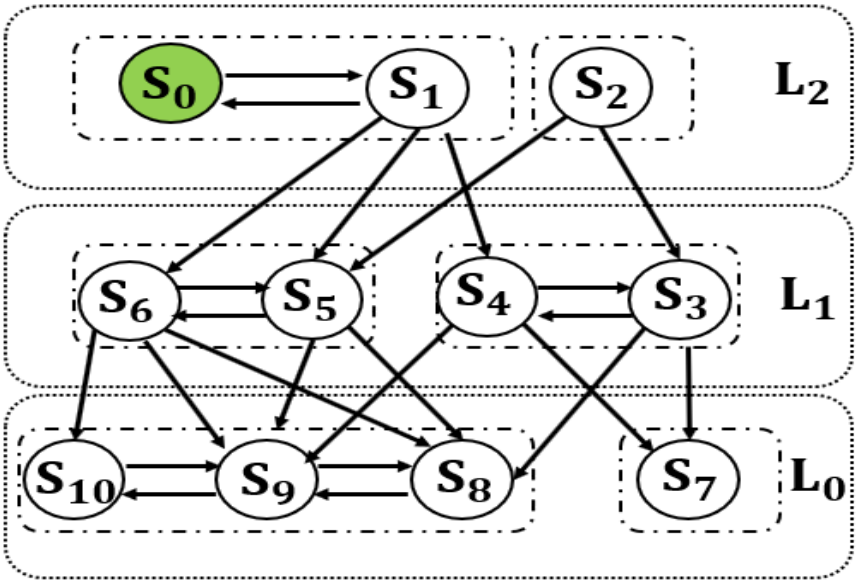

Figure 24 : Exemple d'un PDM, décomposé en niveaux

L'utilité de la décomposition est très puissante, surtout dans les problèmes de très grande taille, vu qu'après avoir décomposé un **PDM** de taille très grande en sous-**PDM**, en niveau, alors on peut traiter chaque niveau indépendamment des autres. Le mot 'indépendance' nous conduit directement à l'utilisation du parallélisme, alors l'utilisation de la version parallèle, proposé de l'algorithme **VI**.

4.4. Distribution Dynamique

Dans cette session nous faire les calculs mathématiques nécessaires, afin de pourrais calculer le nombre du processeur ou thread, qu'on va l'affecter à chaque **CFC**, dans chaque niveau.

Pour faire ces Calculs, nous allons définir les variables suivantes :

$\mathbf{N_L}$:= Nombre des états dans un niveau

$\mathbf{N_P}$:= Nombre des processeurs

$\mathbf{N_{SCC}}$:= Nombre des **SCC** dans un niveau

$\mathbf{N_a}$:= Nombre moyen des processeurs qu'onva affect à chaque **CFC**

Tout d'abord, et en parcourant tous les niveaux, pour chaque niveau on va compter le nombre total des états, dans chaque **CFC**, comme le suivant :

$$\mathbf{N_{CFC}} = \sum_{i=0}^{\text{Nombre d'état}} \mathbf{Etat_i} \qquad (4.1)$$

Maintenant et après avoir compté le nombre d'états, dans chaque **CFC,** on va compter le nombre total des états dans le niveau ou on est situé, en sommant le nombre des états dans chaque **CFC,** de la manier suivante :

$$\mathbf{N_L} = \sum_{\mathbf{k=0}}^{\mathbf{N_{SCC}}} \mathbf{Etat} \qquad \textbf{(4.2)}$$

Notons que cela est fait pour chaque niveau, alors maintenant on le nombre des états dans chaque **CFC**, et aussi le nombre total d'états dans le niveau $\mathbf{L_p}$, il nous reste maintenant que calculer le nombre moyen, avec lequel, on va décider combien de processeur ou thread, on va affecter à chaque **CFC**, dans chaque niveau, en divisant le nombre total des états ($\mathbf{N_L}$), par le nombre des processeurs ($\mathbf{N_p}$).

$$\mathbf{N_a} = \frac{\mathbf{N_L}}{\mathbf{N_p}} \qquad \textbf{(4.3)}$$

Ainsi, on peut décider maintenant, qu'il est le nombre optimal des processeurs, qu'on doit affecter à chaque **CFC**, tout simplement en divisant le nombre des états dans cette composante, par le nombre moyen ($\mathbf{N_a}$), de la manier suivante :

$$\mathbf{N_p} = \frac{\mathbf{N_{CFC}}}{\mathbf{N_a}} \qquad \textbf{(4.4)}$$

Notons que, ce nombre peut avoir une virgule, c'est pour ça qu'on prendre sa partie entière :

$$\mathbf{N_p} = \left[\frac{\mathbf{N_{CFC}}}{\mathbf{N_a}}\right] \qquad \textbf{(4.5)}$$

4.5. Algorithme

L'algorithme, qu'on va présenter, profit à la fois de la version parallèle de l'algorithme **PVI**, ainsi que les algorithmes classiques de la décomposition en niveaux, afin d'avoir la possibilité de résoudre chaque niveau indépendamment, l'algorithme est comme le suivant :

Algorithme 10 : L'algorithme Parallèle Hiérarchique VI

PHVI (MDP: S, A, P, R, N_P : number of processors) *// Parallel Hierarchical VI algorithm*

1. **Construct** Levels using Tarjan's algorithm
2. **For each** $Level\ L_k\ k = 0\ to\ L$ ***Do***

   $N_L \leftarrow 0$ *//Number of states in each level*

   For each $SCC_i\ \ i = 0\ to\ N_{SCC}$ **Do**

   $N_L \leftarrow N_L + N_{SCC_i}$ $\quad // N_{SCC_i}\ is\ Number\ of\ stats\ in\ SCC_i$

   $N_a = \frac{N_L}{N_p}$ //Average number of state in each SCC

   For each $SCC_i\ \ i = 0\ to\ N_{SCC}$ **Do In Parallel**

   $N_{P_i} = \left[\frac{N_{SCC_i}}{N_a}\right]$ $\quad //Number\ of\ processors\ affected\ to\ the\ SCC_i$

   $\boldsymbol{PVI}(MDP_{ik},\ N_{P_i})$

Figure 25 : Présentation de l'algorithme Parallèle Hiérarchique VI

4.6. Comparaison des résultats

Nous avons appliqué cet algorithme sur des **PDM** de grande taille, afin de montrer la performance de l'algorithme **PHVI,** surtout lorsque la taille de la différence, entre les **CFC** devient très grande.

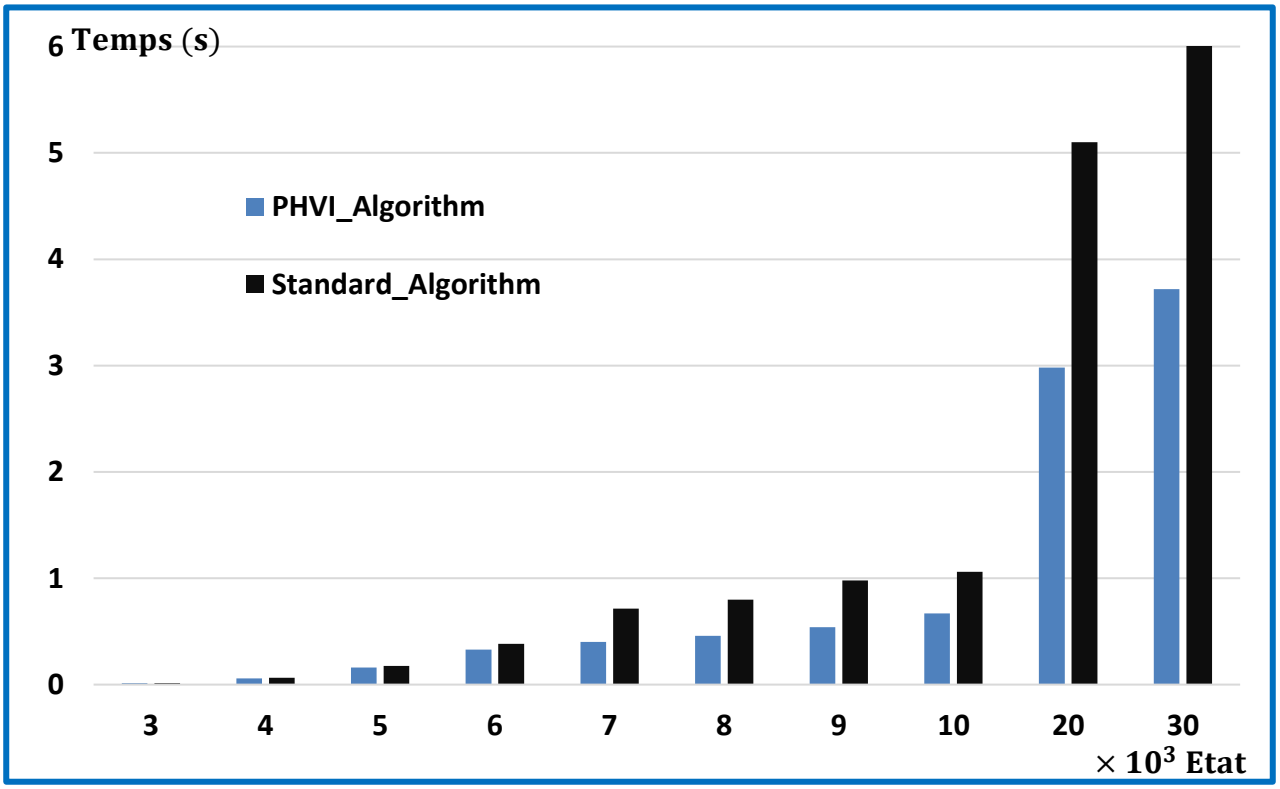

Figure 26 : comparaison entre PHVI et la distribution classique

4.7. Conclusion

Dans ce chapitre, nous avons présenté les versions parallèles des algorithmes, **PVI** et **PMPI**. Après les avoir comparés on peut en déduire ainsi celui qu'on va utiliser dans notre approche. Nous avons profité aussi des techniques de décomposition, en niveaux, pour rendre transformé les **PDM** de grandes tailles, aux **PDM** décomposés en niveaux, afin de résoudre chaque niveau, indépendamment des autres. Ce qui implique l'utilisation du parallélisme à deux niveaux : au niveau de l'algorithme **PVI**, ainsi que, pour la résolution de chaque niveau de l'espace d'états. L'approche proposée a prouvé sa performance, en la comparant avec les méthodes de résolution classiques, et surtout lorsque la différence entre la taille des **CFC** augmente.

Conclusion générale & perspectives

Nous avons présenté dans ce rapport une étude comparative des solveurs **PDM** sous le critère d'actualisation, l'étude était faite selon beaucoup des paramètres, qui sont liés aux problèmes même, ou bien des paramètres liés aux algorithmes, comme le seuil de convergence, ainsi que les comparaisons selon le temps d'exécution, et la complexité. Le but de ce rapport, c'est d'avoir une étude complète, qui va aider à choisir quel algorithme choisi, sur chacun des problèmes qu'on peut affronter, soit ils sont des problèmes de grande taille ou petites tailles. L'étude à contenait aussi, l'analyse des solutions obtenue, qui sont les vecteurs de gains optimaux et les politiques finales, afin d'examiner ces résultats, pour juger aussi quel l'algorithme adéquat pour chaque type de problème.

Le rapport contient aussi une étude théorique, sur ces algorithmes, afin de pouvait juger les critères convergence de ces algorithmes, et savoir déjà s'ils vont converger, quand et sous quelle condition, afin ne pas les appliqués directement sur n'importe quel problème, ou ils peuvent ne pas converger. Alors cette étude, été fait mathématiquement, avec les théorèmes qui assurent l'existence des solutions, ainsi que la convergence des algorithmes, nous avons établit ainsi toutes les preuves de ces théorèmes.

Nous sommes sortis de cette étude avec une nouvelle approche, qui a prouvé sa performance sur les **PDM** de grandes tailles, dans laquelle on a profité à la fois des versions parallèles des algorithmes qu'on a réalisé, ainsi que les méthodes et les algorithmes de composition en niveau, pour construire ainsi, un algorithme nommé **PHVI**, dont on a montré le résultat de sa comparaison avec les méthodes classiques.

Nous allons donc continuer ce travail en cherchant d'autres technique à ajouter dans cette approche, comme la technique d'élimination des actions, afin d'avoir une meilleure optimisation.

Nous continuons donc, la recherche dans le domaine de parallélisme, qu'est un domaine très important, alors, nous allons fait une simulation du parallélisme distribué, en utilisant une plateforme de gestions des machine ou processeurs, qui vont participer à la simulation.

Références :

[1]. Nivot, C. (2016). Analyse et étude des processus markoviens décisionnels (Doctoral dissertation, Bordeaux).

[2]. Analysis of hybrid Systems , (2011). Actes du 5eme Congrès international pluridisciplinaire

[3]. Barnat, J., Chaloupka, J., & Van De Pol, J. (2011). Distributed algorithms for SCC decomposition. Journal of Logic and Computation, 21(1), 23-44.

[4]. Bäuerle, N., & Rieder, U. (2011). Markov decision processes with applications to finance. Springer Science & Business Media.

[5]. Bloemen, V. (2015). On-The-Fly parallel decomposition of strongly connected components (Master's thesis, University of Twente).

[6]. Brunel, N., & Pieczynski, W. (2005). Unsupervised signal restoration using hidden Markov chains with copulas. Signal processing, 85(12), 2304-2315.

[7]. Nivot, C. (2016). Analyse et étude des processus markoviens décisionnels (Doctoral dissertation, Bordeaux).

[8]. Thiéry, C. (2010). Itération sur les politiques optimiste et apprentissage du jeu de Tetris (Doctoral dissertation, Nancy 1).

[9]. Corazza, M. (1975). Techniques mathématiques de la fiabilité prévisionelle des systèmes. Cepadues.

[10]. Matei, Vera. Parallel Algorithms for Detecting Strongly Connected Components. Diss. Vrije Universiteit Amsterdam, 2016.

[11]. Mavard, C. (2017, May). Modélisation par un graphe de flots d'une architecture alternative pour les systèmes de stockage multi-cellulaire de l'énergie électrique. In JCGE (p. 141190).

[12]. Nuel, G., & Prum, B. (2007). Analyse statistique des séquences biologiques: modélisation markovienne, alignements et motifs (p. 200). Hermes.

[13]. Olivier Sigaud, Olivier Buffet , 2008 Processus décisionnels de Markov en intelligence artificielle

[14]. Puterman, M. L. (2014). Markov decision processes: discrete stochastic dynamic programming. John Wiley & Sons.

[15]. Qualité et Sûreté de Fonctionnement QUALITA 2003, Nancy (France), March, 2003a.

[16]. Rabiner, L. R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE, 77(2), 257-286.

[17]. Ross, Sheldon M., et al. Stochastic processes. 1996, Vol. 2. New York: Wiley, 1996.

[18]. Schoenig, R., Aubry, J. F., Cambois, T., & Hutinet, T. (2006). An aggregation method of Markov graphs for the reliability analysis of hybrid systems. Reliability Engineering & System Safety, 91(2), 137-148.

[19]. Schoenig, R., Aubry, J., & Guilhem, E. (2003). al. An example of reliability assessment by an aggregation method of Markov graphs for hybrid systems. In Actes du 5eme Congres international pluridisciplinaire Qualité et Sûreté de Fonctionnement QUALITA 2003.

[20]. Schrodt, P. A. (1999). Early warning of conflict in southern lebanon using hidden markov models. The Understanding and Management of Global Violence, 131-162.

[21]. Soubaras - van den Broek, Hélène & Mattioli, Juliette. (2008). Prévision de l'Évolution future du Risque dans les systèmes Markoviens.

[22]. Tijms, H. C., & Tijms, H. C. (1994). Stochastic models: an algorithmic approach (Vol. 994). Chichester: John Wiley & Sons.

[23]. Wei, W. (2005). Quantifying shared information value in a supply chain using decentralized Markov decision processes with restricted observations. North Carolina State University.